Implementació d'un mòdul de comptadors hardware en el processador OpenRISC 1200

Autor: Albert Batallé García
Grau en Enginyeria Informàtica
Especialitat en Enginyeria de Computadors

Director: Ramon Canal Corretger
Codirector: Roger Espasa Sans
Departament d'Arquitectura de Computadors (DAC)

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) – Barcelona Tech

Dijous 6 de febrer de 2014

RESUM

Els comptadors hardware són uns registres interns al processador que compten esdeveniments. D'aquesta manera es pot recopilar informació del que està passant internament al processador.

Després d'una selecció sobre diferents processadors existents, s'analitza i es genera documentació per al processador OR1200 basat en OpenRISC. Finament s'implementa un mòdul de comptadors hardware en el processador i es presenten resultats del correcte funcionament de la implementació basant-se en uns jocs de proves.

RESUMEN

Los contadores hardware son unos registros internos al procesador que cuentan eventos. De esta forma se recopila información de lo que está sucediendo internamente al procesador.

Tras una selección sobre diferentes procesadores existentes, se analiza y genera documentación para el procesador OR1200 basado en OpenRISC. Finalmente se implementa un módulo de contadores hardware en el procesador y se presentan resultados del correcto funcionamiento de la implementación basándose en unos test.

ABSTRACT

Performance counters are internal registers in the processor that count events. Thereby information about what is happening in the core is collected.

After selecting a processor from a variety of them, documentation for OR1200 processor, which is based on OpenRISC, has been analyzed and generated. Lastly, a performance counters module has been implemented in the processor and results of the implementation correctness are provided through executed tests.

AGRAÏMENTS

El TFG és un treball de caràcter personal però altres persones resulten implicades i és humà reconèixer el seu ajut.

En primer lloc vull agrair al director del projecte, Ramon Canal Corretger, per acceptar ser-ne el director i la paciència en la lectura dels correus electrònics extensos que li enviava.

En segon lloc al codirector, Roger Espasa Sans, per fer realitat el projecte al proposar afegir un mòdul de comptadors hardware en un processador que s’havia d’escollir.

Finalment, al professor Josep-Llorenç Cruz Díaz, per rebre’m quan no ens podíem trobar amb el director i per la resolució de petits dubtes sorgits tot i no participar d'aquest projecte.

ÍNDEX DE CONTINGUTS

1 INTRODUCCIÓ 1

1.1 MOTIVACIÓ 1

1.2 OBJECTIUS 1

1.3 ELS COMPTADORS HARDWARE 2

1.4 RELLEVÀNCIA I JUSTIFICACIÓ 3

1.5 METODOLOGIA 3

1.6 DOCUMENTACIÓ OR1200 GENERADA 5

2 ESTAT DE L'ART 6

2.1 LLENGUATGES DE DESCRIPCIÓ DE HARDWARE (HDL) 6

2.2 PROCESSADORS EXISTENTS 7

2.3 COMPTADORS HARDWARE ACTUALS 9

3 OPENRISC 11

3.1 QUÈ ÉS OPENRISC 11

3.2 L'ARQUITECTURA OPENRISC 1000 11

3.2.1 El conjunt de registres 12

3.2.2 El joc d'instruccions 13

3.2.2.1 El format de les instruccions 14

3.3 ORPSOC 17

3.4 EL PROCESSADOR OR1200 19

3.4.1 El pipeline 21

4 EINES I RECURSOS SW I HW 25

4.1 PLACA DE1 D'ALTERA 25

4.1.1 Les FPGA 26

4.2 SOFTWARE QUARTUS II D'ALTERA 29

4.3 SOFTWARE LOGICWORKS 29

4.4 SIMULADOR MODELSIM 30

4.5 SIMULADOR ICARUS VERILOG + VISUALITZADOR GTKWAVE 30

4.6 TOOL CHAIN D'OPENRISC 31

4.6.1 Simulador OR1ksim ... *32*
4.6.2 Imatge de Linux ... *32*
4.7 OR1K-TCLTOOLS ... 32
4.8 EINES DE L'ORPSOC ... 33
4.8.1 Obtenir el fitxer binari per a l'FPGA ... *34*
4.8.2 Programar el processador a l'FPGA ... *34*
4.8.3 Simular el processador ... *35*
5 DISSENY I IMPLEMENTACIÓ DELS COMPTADORS HARDWARE ... 36
5.1 ESDEVENIMENTS ... 36
5.1.1 Definició dels esdeveniments ... *37*
5.2 UBICACIÓ DEL MÒDUL PCU ... 39
5.3 REGISTRES DEL MÒDUL PCU ... 40
5.3.1 Registres de configuració, grup SPR 0 ... *40*
5.3.2 Registres PCCR0-PCCR7 ... *42*
5.3.3 Registres PCMR0-PCMR7 ... *42*
5.4 LA LÒGICA DEL MÒDUL PCU ... 43
5.4.1 Implementació dels esdeveniments ... *44*
5.4.2 Implementació dels registres comptadors ... *47*
5.5 IMPLEMENTACIÓ D'UN VISUALITZADOR DE COMPTADORS EXTERN AL PROCESSADOR ... 51
6 RESULTATS ... 53
6.1 IMPACTE DEL BUS I DE LA MEMÒRIA PRINCIPAL EN ELS RESULTATS ... 59
7 DESENVOLUPAMENT DEL PROJECTE ... 60
7.1 PLANIFICACIÓ ... 60
7.1.1 Planificació inicial ... *60*
7.1.2 Canvi de planificació ... *62*
7.1.3 Planificació real ... *63*
7.2 COST DEL PROJECTE ... 64
7.3 SOSTENIBILITAT I COMPROMÍS SOCIAL ... 66
7.4 LLEIS I REGULACIONS ... 67
7.5 COMPETÈNCIES TÈCNIQUES ... 68
8 CONCLUSIONS ... 69

9 BIBLIOGRAFIA **70**

ANNEX A MATERIAL ADJUNT **72**

ANNEX B COMPTADORS A INTEL I ARM **73**

ANNEX C CIRCUITS D'OPENRISC **76**

ANNEX D JOC DE PROVES: PCU **89**

D.1 CODI DEL JOC DE PROVES: 89

D.2 DESCRIPCIÓ DEL JOC DE PROVES 109

ANNEX E GUIA D'ERRORS **114**

E.1 MENTRE ES PROGRAMA LA PLACA 114

E.2 SIMULANT AMB MODELSIM 114

ÍNDEX DE TAULES

TAULA 2-1. RELACIÓ DE PROCESSADORS 8

TAULA 3-1. GRUPS PRINCIPALS DELS REGISTRES DE PROPÒSIT ESPECIAL (SPR) 13

TAULA 3-2. CODI MÀQUINA DE LES INSTRUCCIONS ORBIS32 16

TAULA 5-1. ESDEVENIMENTS DEFINITS PEL MANUAL D'ARQUITECTURA OR1K 36

TAULA 6-1. JOC DE PROVA ESDEVENIMENT IF 54

TAULA 6-2. JOC DE PROVA ESDEVENIMENT ICM 55

TAULA 6-3. JOC DE PROVA ESDEVENIMENT ICM 56

TAULA 6-4. JOC DE PROVA ESDEVENIMENTS LA I SA AMB COMPROVACIÓ DEL BIT CIUM 57

TAULA 6-5. JOC DE PROVA ESDEVENIMENT BS 58

TAULA 7-1. COST TOTAL DEL PROJECTE 66

ÍNDEX D'IL·LUSTRACIONS

IL·LUSTRACIÓ 1-1. DIAGRAMA DE PASSOS A SEGUIR 4
IL·LUSTRACIÓ 2-1. REGISTRE EN VERILOG I VHDL 7
IL·LUSTRACIÓ 3-1. COMPONENTS ORPSOCV2 PER A LA PLACA DE1 18
IL·LUSTRACIÓ 3-2. PROCESSADOR OR1200: MÒDUL “OR1200_TOP” 19
IL·LUSTRACIÓ 3-3. JERARQUIA DE MEMÒRIA OR1200 20
IL·LUSTRACIÓ 3-4. CIRCUIT DEL FITXER OR1200_OPERANDMUXES.V 22
IL·LUSTRACIÓ 3-5. PIPELINE DEL PROCESSADOR OR1200 23
IL·LUSTRACIÓ 4-1. PLACA DE1 D’ALTERA 25
IL·LUSTRACIÓ 4-2. CONFIGURACIÓ D’UNA LUT D’EXEMPLE 27
IL·LUSTRACIÓ 4-3. ESQUEMA DE LA INTERCONNEXIÓ D’UNA FPGA 28
IL·LUSTRACIÓ 4-4. SOFTWARE LOGICWORKS 29
IL·LUSTRACIÓ 4-5. VISUALITZADOR DE SENYALS GTKWAVE 31
IL·LUSTRACIÓ 5-1. PROCESSADOR OR1200: MÒDUL “OR1200_TOP” AMB PCU 40
IL·LUSTRACIÓ 5-2. DESCRIPCIÓ DELS BITS DEL REGISTRE PCCFGR (PCCFGR FIELD DESCRIPTIONS) 41
IL·LUSTRACIÓ 5-3. REGISTRE PCCR 42
IL·LUSTRACIÓ 5-4. REGISTRE PCMR 43
IL·LUSTRACIÓ 5-5. INTERFÍCIE PCU 44
IL·LUSTRACIÓ 5-6. CRONOGRAMA D’ESDEVENIMENT DE FALLADES DE TLB 46
IL·LUSTRACIÓ 5-7.PIPELINE DURANT UN SALT 47
IL·LUSTRACIÓ 5-8. CIRCUIT LÒGIC DEL MÒDUL PCU 50
IL·LUSTRACIÓ 5-9. CONNEXIÓ PCU_IO_CONTROL AMB LA RESTA D’ELEMENTS 51
IL·LUSTRACIÓ 5-10. DEMOSTRACIÓ DELS CASOS DESCRITS (PCU_IO_CONTROL) 52
IL·LUSTRACIÓ 6-1. CAPTURA DE PANTALLA D'UNA SIMULACIÓ 53
IL·LUSTRACIÓ 7-1. DIAGRAMA DE GANTT INICIAL 60
IL·LUSTRACIÓ 7-2. DIAGRAMA DE GANTT (SETEMBRE) 62
IL·LUSTRACIÓ 7-3. DIAGRAMA DE GANTT DE LA PLANIFICACIÓ REAL 64
IL·LUSTRACIÓ B-1. REGISTRE IA32_PERFEVTSELX D’INTEL 74
IL·LUSTRACIÓ B-2. REGISTRE PMSELR D’ARM 75
IL·LUSTRACIÓ C-1. CIRCUIT LÒGIC MÒDUL OR1200’S GENERATE PC 76
IL·LUSTRACIÓ C-2. CIRCUIT LÒGIC MÒDUL OR1200'S INSTRUCTION FETCHER 77
IL·LUSTRACIÓ C-3. CIRCUIT LÒGIC MÒDUL OR1200'S REGISTER FILE 78
IL·LUSTRACIÓ C-4. CIRCUIT LÒGIC MÒDUL OR1200'S REGISTER FILE READ OPERANDS MUX 79
IL·LUSTRACIÓ C-5. CIRCUIT LÒGIC MÒDUL OR1200'S WRITE-BACK MUX 79
IL·LUSTRACIÓ C-6. CIRCUIT LÒGIC MÒDUL OR1200'S LOAD / STORE UNIT (EX STAGE) 80
IL·LUSTRACIÓ C-7. CIRCUIT LÒGIC MÒDUL OR1200'S LOAD / STORE UNIT (ID STAGE) 81
IL·LUSTRACIÓ C-8. CIRCUIT LÒGIC MÒDUL OR1200'S VR, UPR AND CONFIGURATION REGISTERS (ABANS DE PCU) 81

IL·LUSTRACIÓ C-9. CIRCUIT LÒGIC MÒDUL OR1200'S FREEZE LOGIC 82
IL·LUSTRACIÓ C-10. CIRCUIT LÒGIC MÒDUL OR1200'S INSTRUCTION DECODE (IF STAGE) 83
IL·LUSTRACIÓ C-11. CIRCUIT LÒGIC SENYAL 'NO_MORE_DSLOT' MÒDUL OR1200'S INSTRUCTION DECODE (ID STAGE) 83
IL·LUSTRACIÓ C-12. CIRCUIT LÒGIC MÒDUL OR1200'S INSTRUCTION DECODE (EX STAGE) 84
IL·LUSTRACIÓ C-13. CIRCUIT LÒGIC MÒDUL OR1200'S INSTRUCTION DECODE (WB STAGE) 84
IL·LUSTRACIÓ C-14. CIRCUIT LÒGIC MÒDUL OR1200'S DATA CACHE TOP LEVEL 85
IL·LUSTRACIÓ C-15. CIRCUIT LÒGIC MÒDUL OR1200'S INSTRUCTION CACHE TOP LEVEL 86
IL·LUSTRACIÓ C-16. CIRCUIT LÒGIC MÒDUL OR1200'S DATA MMU TOP LEVEL 87
IL·LUSTRACIÓ C-17. CIRCUIT LÒGIC MÒDUL OR1200'S INSTRUCTION MMU TOP LEVEL 88

ABREVIATURES I SÍMBOLS

A continuació es pot trobar una llista d'abreviatures i acrònims utilitzats. Al llarg de la memòria s'han fet servir anglicismes que s'han marcat amb cursiva. S'ha considerat deixar noms en anglès perquè són paraules conegudes àmpliament al món informàtic.

Els acrònims dels esdeveniments dels comptadors es troben a la Taula 5-1 i, per tant, aquí no s'han replicat.

ACK	Acknowledgement
ASIC	Application Specific Integrated Circuit / Circuits integrats d'aplicació específica
CPUCFGR	CPU Configuration Register / Registre de configuració de la CPU
DC	Data Cache / Cache de dades
DE1	Development & Education board / Placa física utilitzada
DMMU	Data Memory Management Unit / Unitat de gestió de memòria de dades
DU	Debug Unit / Unitat de debug
EX	Execution stage / Etapa d'execució
FPGA	Field Programmable Gate Array
GNU	GNU's Not Unix
GPL	GNU General Public License / Llicència pública general de GNU
GPR	General Purpose Registers / Registres de propòsit general
HDL	Hardware Description Language / Llenguatge de descripció de hardware
HW	Hardware / Maquinari
IC	Instruction Cache / Cache d'instruccions
ID	Instruction Decode stage / Etapa de descodificació
IF	Instruction Fetch stage / Etapa de cerca
IMMU	Instruction Memory Management Unit / Unitat de gestió de memòria d'instruccions
IP Core	Intellectual Property Core
JTAG	Join Test Action Group / Port de connexió
LE	Logic Element / Composició d'una FPGA
LSU	Load Store Unit / Unitat de Loads i Stores
LUT	Lookup Table / Part d'una FPGA
MAC	Multiply and Accumulate unit / Unitat multiplicadora i acumuladora
MEM	Memory stage / Etapa de memòria
MSR	Model Specific Registers / Registres especials d'Intel
OR1200	OpenRISC 1200 Processor / Processador d'OpenRISC
OR1K	Identificador d'OpenRISC 1000

ORPSoC	OpenRISC Reference Platform SoC / SoC de referència d’OpenRISC
PC	Program Counter / Registre comptador de programa
PCCFGR	Performance Counters Configuration Register / Registre de configuració dels comptadors hardware
PCCR	Performance Counter Count Register
PCMR	Performance Counter Mode Register
PCU	Performance Counters Unit / Unitat de comptadors hardware
PIC	Programmable Interrupt Controller / Controlador programable d’interrupcions
PM	Power Management / Unitat de control de consum
RTL	Register-Transfer level / Nivell d’abstracció en HDL
SoC	System on Chip / Sistema en un xip
SPR	Special Purpose Registers / Registres de propòsit especial
SW	Software / Programari
TLB	Translation Lookaside buffer
TT	Tick Timer / Unitat de rellotge
UPR	Unit present Register / Registre d’unitats presents
VCD	Value Change Dump file / Fitxer de bolcatge
VHDL	Very high speed integrated circuit HDL
VR	Version Register / Registre de versió
WB	Write Back stage / Etapa d’escriptura

1 INTRODUCCIÓ

1.1 Motivació

Les eines software han evolucionat ràpidament. L'anàlisi d'aplicacions i la seva correctesa poden ser comprovades amb simulacions, tests exhaustius i eines de depuració. Però quan ens traslladem al món hardware, aquests mateixos procediments solen complicar-se i arribar a l'extrem en què els xips són com caixes negres, on desxifrar el que hi està passant no resulta ser una tasca senzilla.

Així doncs, és de gran interès obtenir informació de primera mà de dins d'un processador: quin és l'estat de la CPU en cada instant, quines senyals s'activen, els esdeveniments que estan succeint internament i, d'aquesta manera, poder precisar com és l'execució d'un codi en el processador.

Els "comptadors hardware" o altrament coneguts com "*performance counters"* exploren aquest món i són l'objectiu d'aquest projecte. Obtenir informació privilegiada sense la complexitat d'una gran implementació envers altres sistemes és un dels principals atractius. Nascuts com a mode de depuració pels mateixos fabricants de processadors i que passaven desapercebuts per la resta als seus inicis, s'han anat estenent en tot l'àmbit de desenvolupament, fent-los populars.

1.2 Objectius

El present projecte s'emmarca dins d'un entorn acadèmic i, per tant, el seu principal objectiu és el d'estendre les capacitats i configuració d'un processador per a l'ús docent.

Així doncs, primer caldrà avaluar qualitativament diferents processadors disponibles per FPGA per poder seleccionar aquell que més convingui.

Posteriorment, s'ampliarà aquest processador amb comptadors hardware. En concret s'analitzarà el codi font i s'obtindran esquemes del circuit a nivell lògic i s'afegirà el mòdul de comptadors hardware a nivell d'implementació.

1.3 Els comptadors hardware

Els comptadors hardware són uns registres específics que compten esdeveniments que tenen lloc dins del processador durant l'execució d'un programa. Aquests registres són limitats i depenen de la implementació del processador, possibilitant que varïi la seva especificació en cada versió, és a dir, que no siguin necessàriament compatibles amb arquitectures anteriors o posteriors.

Els registres que formen els comptadors solen dividir-se en dos tipus segons les seves característiques: un primer registre de configuració per establir quan comptar i quins esdeveniments comptar (com a paràmetres destacables), vinculats a un segon registre que és el propi comptador, el que emmagatzema el valor i es va incrementant.

Al ser uns registres específics segons la implementació, formen part dels registres de propòsit especial, o en nomenclatura d'Intel, MSR (*Model Specific Registers)* i són accessibles amb les instruccions pròpies de lectura o escriptura a registres especials.

Els esdeveniments que es poden comptar vénen definits per la pròpia especificació però els més comuns són accessos o fallades a TLB, a memòria cache de nivell 1 o 2, nombre d'instruccions portades, salts...

Com que el resultat d'aquests esdeveniments és una informació molt útil per a un anàlisi de rendiment d'un codi després d'una execució, aquests comptadors també s'anomenen comptadors de rendiment, on és més estès el nom en anglès: *performance counters.*

1.4 Rellevància i Justificació

Aquest projecte és un projecte indicat especialment per a l'especialitat d'Enginyeria de Computadors, ja que es tracta de modificar en part i ampliar un processador definit en llenguatge de descripció de hardware (HDL).

En certa manera, dues assignatures de l'especialitat hi són molt presents. La més directa, PEC (Projecte d'Enginyeria de Computadors) ja que consisteix essencialment en la mateixa dinàmica de classe, però amb un processador diferent i implementant una part no vista en aquella assignatura.

L'altra assignatura és AC2 (Arquitectura de Computadors II) on es treballa a nivell teòric un processador segmentat i l'estudi de les etapes de segmentació, amb els inherents avantatges i problemes que sorgeixen. Haver obtingut aquests coneixements és vital per treballar sobre un processador segmentat i entendre el seu funcionament.

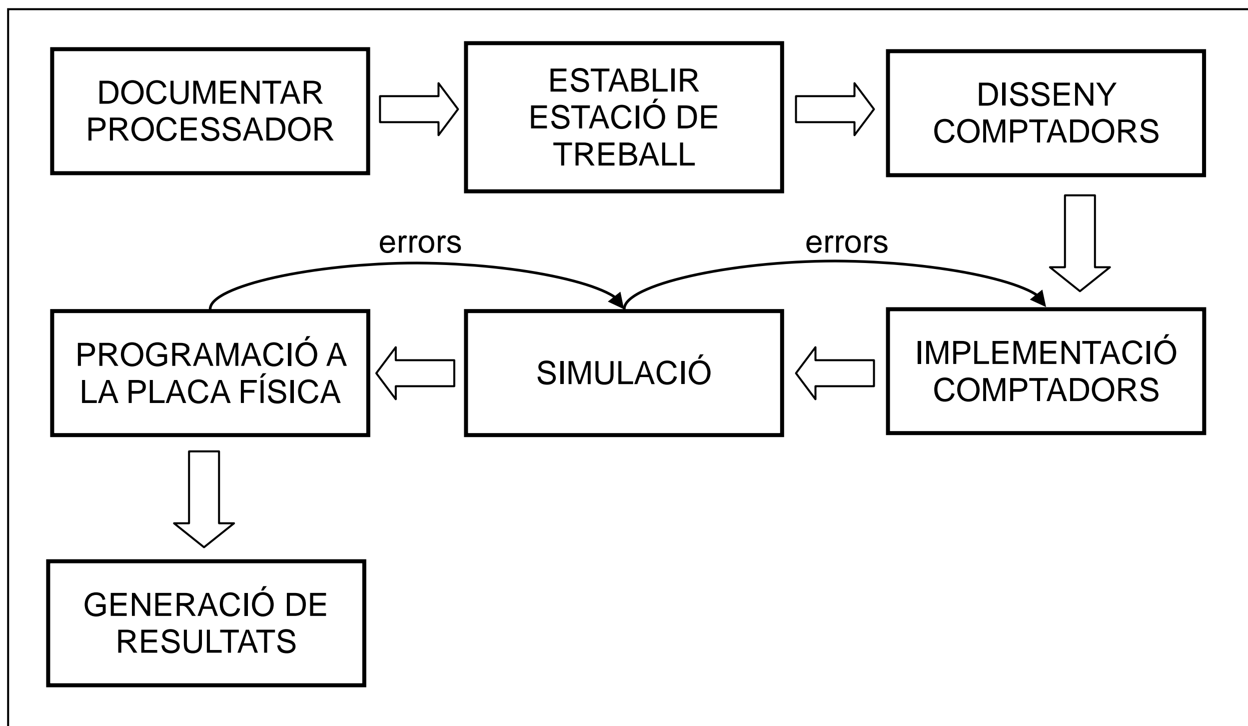
D'altra banda, es pot comprovar el comportament d'aquest processador sobre un dispositiu hardware com podria ser una FPGA (*Field Programable Gate Array*) sense necessitat de disposar d'un xip específic. Aquest fet és molt útil per dedicar-se a la programació i testeig en un entorn acadèmic.

Finalment, el perquè d'aquest projecte és també de caire personal. Aprendre i aprofundir en el disseny, creació o comportament d'un processador és una forma excel·lent d'ampliar i assegurar coneixements, per a un possible futur laboral.

1.5 Metodologia

Primer s’han avaluat qualitativament diferents processadors disponibles per FPGA per poder seleccionar aquell que s’adaptava millor al propòsit del projecte (apartat 2.2).

Després se seguiran els passos que es mostren esquemàticament a la Il·lustració 1-1 i que es detallen a continuació.



Il·lustració 1-1. Diagrama de passos a seguir

El punt de partida inicial és el de documentar el processador i conèixer el seu comportament. Per fer això, serà necessari disposar del codi font, analitzar-lo i dibuixar el circuit que el compon per a una fàcil comprensió.

El codi font del processador escollit (OpenRISC 1200) està estructurat en mòduls i no pas dividit per etapes de segmentació. Un diagrama de la configuració del processador segmentat sol ser d'interès, doncs amb un cop d'ull es veu per quines etapes passarà i quin és el camí de dades i les parts més importants. Així doncs, obtenir un diagrama del *pipeline* del processador serà un dels primers passos.

Un cop obtinguda tota aquesta primera part de documentació, es descarregaran les eines a utilitzar i es provaran. En aquest cas, sabem que volem simular el seu comportament, per tant, necessitarem un simulador. El processador ja ve amb tot un conjunt d'eines que s'explicarà més endavant, les quals també seran necessàries instal·lar per la seva manipulació.

Sabent això, es procedirà a la inserció del mòdul de comptadors hardware en aquest processador. Primer es farà un disseny sobre el mòdul dels comptadors amb les senyals i components necessaris, i també es detectarà les parts implicades al processador com puguin ser els mòduls i senyals existents que s'hagin de modificar o interconnectar.

A continuació, es procedirà a la implementació dels comptadors. Es modificarà el codi font afegint aquesta nova funcionalitat i per assegurar el correcte funcionament i que no s'està malmetent el processador, cada cop que s'arribi a un punt estable, és a dir, cada cop que el codi sigui compilable de nou i l'execució no causaria problemes encara que no donés resultats correctes (per estar en una fase intermèdia), es provaran uns jocs de prova que segueixin assegurant el correcte comportament del processador. Així doncs, se seguirà un procediment d'implementació – simulació – prova a la placa fins a obtenir el mòdul final i que es garanteixi el funcionament del processador sencer de nou.

En aquesta última fase, es provaran uns jocs de proves molt complerts que ofereix el mateix processador, que proven els diferents mòduls existents. Si són satisfactoris, significarà que el processador segueix funcionant com a l'estat inicial. Seguidament es provaran uns codis creats específicament per demostrar el funcionament dels comptadors, vinculant els resultats obtinguts amb el previ anàlisi teòric.

1.6 Documentació OR1200 generada

El projecte consta de tot un seguit de taules, material i il·lustracions que han sigut creades específicament per a la realització del projecte i no han estat extretes de terceres parts.

En concret, totes les il·lustracions, diagrames i taules del capítol 3 han estat d’elaboració pròpia, exceptuant la Taula 3-1 que procedeix del propi manual de l’arquitectura OpenRISC.

Així mateix, l’Annex C, que conté tot un seguit de circuits creats amb el software Logic Works i que no provenen de cap font externa, també ha estat elaborat íntegrament per a la realització d’aquest projecte.

2 ESTAT DE L'ART

Per acomplir el primer objectiu és necessari disposar d'un processador per a utilitzar-lo en l'àmbit acadèmic. Així doncs, cal descobrir què existeix per tal de poder fer una elecció.

El que es vol exactament és un processador bastant complert que pugui funcionar sobre una FPGA (element que es descriu al capítol 4) escrit en llenguatge HDL.

2.1 Llenguatges de descripció de hardware (HDL)

Els llenguatges de descripció de hardware (HDL) [1] són llenguatges que defineixen el comportament i la temporització dels components electrònics, és a dir, faciliten una forma per descriure el hardware.

Podrien ser equiparables als llenguatges de programació d'alt nivell per al software, sent els cables i senyals les variables, però no seria exactament correcte. Els components electrònics, a nivell físic, requereixen de més aspectes que no una programació software perquè depenen d'un concepte clau: el temps.

Així doncs, els HDLs permeten la parametrització temporal de les variables i la concurrència d'instruccions i assignacions. Això és necessari perquè en un circuit real, totes les senyals tenen sempre un valor, i poden canviar o actualitzar-se alhora en un mateix instant. És per això que una instrucció escrita posterior a una altra en HDL no significa que s'executi més tard que l'altra (de forma seqüencial), sinó que poden fer-ho alhora o segons el moment temporal assignat.

Actualment, els dos llenguatges HDL més estesos i acceptats són Verilog i VHDL. En realitat, per a l'objectiu del projecte és indiferent amb quin dels dos llenguatges estigui programat el processador base, doncs les diferències més destacables són la sintaxi a utilitzar i canviar d'un llenguatge a un altre no és una tasca realment complicada, si es té prèvia noció d'algun llenguatge HDL. En la següent figura es pot veure la comparativa entre definir un registre en Verilog i en VHDL.

Registre en Verilog:	Registre en VHDL:

```verilog
module flipflop
   (clk, D, Q);

input  clk;
input  D;
output Q;
reg    Q;


always @(posedge clk)
    begin
        Q <= D;
    end
endmodule
```

```vhdl
LIBRARY IEEE;
USE ieee.std_logic_1164.all;

Entity flipflop IS
    port(D :    in std_logic;
         clk :  in std_logic;
         Q :    out std_logic
    );
End flipflop;

ARCHITECTURE behavior OF flipflop IS
BEGIN
    process (D, clk)
    begin
        if(clk'event and clk='1') then
            Q <= D;
        end if;
    end process;
END behavior;
```

Il·lustració 2-1. Registre en Verilog i VHDL

Definint el hardware, també existeixen diferents nivells d'abstracció. El més comú i amb el que es buscarà el codi és el nivell RTL (*Register-transfer level*). Aquest nivell descriu el comportament del circuit, però no necessàriament representa les portes bàsiques (or, and, not...) amb el que es traduirà finalment el codi un cop sigui "sintetitzat".

Altres nivells d'abstracció són el de comportament (*behavioral)* i a nivell de portes (*gate level)*. El nivell de portes és definir tota la lògica amb portes bàsiques, que és equivalent a "sintetitzar" el codi RTL. El nivell de comportament, per contra, és el major nivell d'abstracció, indicant la funcionalitat del disseny però podent no ser sintetitzable. La Il·lustració 2-1 és escrita amb el nivell RTL.

2.2 Processadors existents

Quan s'ha parlat que es vol un processador bastant complet, significa que es vol, a ser possible, una versió SoC amb *IP Cores* integrats, per tal d'agilitzar la feina i centrar-se només als objectius del projecte.

Un SoC (*System on a Chip)* és el concepte d'integrar en un xip diferents components d'un sistema. Així doncs, en aquest xip no només contindria el processador en si mateix o la

CPU, sinó també el controlador de memòria, de pantalla, els busos, comunicació, etc. Aquests altres elements, com els busos, el controlador de pantalla, la connexió UART, etc., són el que s'anomenen *IP Cores*, de l'anglès *Intellectual Property Core*. Són codis reutilitzables i portables per estalviar el disseny de nou d'un mòdul en concret i poder-lo emprar.

Així doncs, cal buscar quins processadors SoC existeixen escrits en RTL – HDL, i les seves característiques. La Taula 2-1 mostra una comparació de diferents processadors trobats.

Processador	Arquitectura	Bits	HDL	Llicència	DOCS	SO i Drivers
Cortex-M1 [2]	Armv6-M	32	Codi no visible, protegit	Propietari	Sí	µC/OS-II (no MMU)
ZET [3]	X86 (IA-32)	16	Verilog	GPL	No	DOS i Windows
OpenRISC [4]	OR1K	32 o 64	Verilog	GPL	Sí. Fòrum comunitat activa	Linux, (yCLinux), Wishbone interface, *cores* per PS/2, VGA… a OpenCores
LEONv3 [5]	SPARCv8	32	VHDL	GPL	Sí	SnapGear embedded system, Linux
S1 Core [6]	SPARCv9 (single core UltraSPARC T1)	64	Verilog	GPL	Especificació obsoleta	Wishbone interface, Linux kernel
MIPS	MIPS	16 o 32	Verilog o VHDL	Diverses	Hi ha diferents tipus de MIPS, com el MIPS R2000 [7]	

Taula 2-1. Relació de processadors

A la taula, seguint les columnes d'esquerra a dreta, es descriu per cada processador la seva d'arquitectura; el nombre de bits de l'arquitectura, indicant el nombre de bits de les dades i dels registres; en quin llenguatge HDL està escrit; sota quin tipus de llicència es pot utilitzar el processador; si existeix documentació sobre l'especificació de l'arquitectura, el funcionament o la implementació del processador; i finalment *cores* existents per al processador i tipus de sistema operatiu que pot córrer.

La llicència GPL és gratuïta, de codi lliure, i no hi ha problema en utilitzar el processador sempre i quan es reconeguin i mantinguin els autors. El *Wishbone interface* es tracta del bus d'interconnexió comú entre tots els *cores* i el processador.

El criteri de selecció ha estat que fos de codi lliure (llicència GPL), que existeixi prèviament documentació i si disposa de *cores* és un factor a tenir en compte respecte un que no en tingui.

Així doncs, el S1-Core i el Cortex-M1 s'han descartat d'entrada. El Zet, tot i no tenir documentació, estava comprovat que funciona sobre la placa física que utilitzarem i realment era un atractiu. Al final però, s'ha descartat degut a les micro-instruccions de l'arquitectura d'Intel, més complexa respecte a d'altres arquitectures. Dels 3 restants, el LEONv3 semblava disposar de menys característiques que els altres dos. La selecció final doncs, depèn entre seleccionar entre un MIPS i l'OpenRISC.

S'ha valorat la possibilitat del MIPS, però s'ha arribat a la conclusió que pot ser innovador per als estudiants si s'utilitza un altre tipus de processador que no hagi estat utilitzat abans en l'entorn docent, perquè així pot donar la sensació d'evolució i no de repetir tasques prèvies i coneixements.

Així doncs, i amb els directors del projecte d'acord, s'ha establert que OpenRISC és la opció amb la que es prosseguirà, el que s'utilitzarà com a processador base. Les seves característiques a grans trets són: és de codi lliure, disposa de IP *Cores* per a tots els components (pantalla, teclat, USB, bus *Wishbone*, etc.), hi ha tota una comunitat que manté actiu el projecte d'OpenRISC, utilitza una arquitectura de 32 bits RISC, està escrit amb llenguatge Verilog i internament consta d'un disseny segmentat en etapes.

Al capítol 3 es descriu amb més profunditat l'arquitectura i el processador basat en OpenRISC.

2.3 Comptadors hardware actuals

Al capítol d'introducció s'ha definit què és un comptador hardware i es pot entreveure que no és un element gaire complicat i que el seu ús en tasques d'anàlisi de rendiment està creixent. Tot fa pensar doncs, que si és senzill i avantatjós, ja estigui implementat. És cert. Existeixin diferents versions a la xarxa, com aquest mòdul de comptadors hardware per al processador *MicroBlaze* de Xillinx [8].

Les companyies Intel i ARM, en els seus processadors també inclouen aquests comptadors. No obstant, cadascú els ha implementat amb criteris diferents (quan es compten i com es compten), però basant-se amb el mateix concepte: uns registres de configuració per a comptar uns esdeveniments. En l'Annex B, es mostra el registre de configuració d'aquests comptadors tant d'una empresa com de l'altre, extretes del seu manual d'arquitectura, respectivament [9, 10].

Així doncs, per què no s'utilitza un mòdul ja implementat? Doncs exactament per aquest mateix fet que ja s'ha descrit. A nivell conceptual, els comptadors són exactament iguals a la gran majoria de versions existents: Tan sols és necessari disposar d'uns registres que comptin, dels registres que configurin aquests comptadors i dels esdeveniments que es vulguin comptar. Però a l'hora d'implementar-los físicament, cap de les versions existents és d'utilitat.

No es poden utilitzar, perquè la seva implementació depèn totalment de l'arquitectura del processador. Pel simple fet de comptar uns determinats esdeveniments, comporta que s'hagin de connectar components i mòduls entre si que abans no necessàriament haurien d'estar connectats per tal de ser conscients d'aquell esdeveniment i poder-lo capturar (i fins hi tot, pot provocar canvis entre diferents versions d'un mateix processador). Tal com diuen Intel o ARM als seus manuals, la implementació és depenent sense necessitat que sigui compatible amb altres versions del processador.

És a dir, en aquest sentit es podrà seguir el concepte i la descripció teòrica d'un mòdul genèric de comptadors, però es requerirà una implementació completament nova per a aquest processador OpenRISC, el qual, no té implementat el mòdul, però en el manual de la seva arquitectura sí que està definida l’especificació dels comptadors i dels esdeveniments a utilitzar, com veurem al capítol 5.

3 OPENRISC

3.1 Què és OpenRISC

OpenRISC [4] és un dels projectes principals d'*OpenCores*, una comunitat dedicada a la lliure distribució i implementació de hardware [11].

El projecte es compon de tota una plataforma, partint d'una especificació de l'arquitectura, però també d'un conjunt de processadors que implementen l'arquitectura, versions *SoC* i un conjunt d'eines software com simuladors, compiladors..., i alguns sistemes operatius que es poden executar sobre aquesta arquitectura.

També s'han creat implementacions comercials, terceres empreses l'han utilitzat en algun dels seus productes [12], i s'ha fabricat una placa específica d'OpenRISC per al seu desenvolupament [13].

3.2 L'arquitectura OpenRISC 1000

L'arquitectura OpenRISC 1000, coneguda per OR1K [14], és el primer conjunt d'especificacions per a la família de processadors RISC de 32 i 64 bits, recollides en el primer manual de l'arquitectura.

L'arquitectura defineix el joc d'instruccions, el conjunt de registres, els models de memòria i d'excepcions, el mode d'adreçament (no es permeten accessos no alineats), incorpora un *delay slot* per mantenir el pipeline amb instruccions durant un salt i dóna suport per al canvi de context als registres, caches i MMUs ràpid.

Totes les especificacions es poden trobar al manual de l'arquitectura, però el format específic de cada component o el model utilitzat ve marcat per la implementació realitzada sobre aquesta especificació.

3.2.1 El conjunt de registres

L'arquitectura defineix 32 registres de propòsit general (GPR) de 32 o 64 bits, anomenats R0..R31. En implementacions de baix consum es poden utilitzar únicament els 16 primers, però això ve determinat per als criteris d'implementació seguits.

No obstant, del llistat d'aquests registres alguns tenen una funcionalitat específica:

- El registre R0 és utilitzat com a zero constant i mai pot ser utilitzat com a registre destinació.
- El registre R1 és utilitzat com a *Stack Pointer* (SP).
- El registre R2 és utilitzat com a *Frame Pointer* (FP).
- Del registre R3 al R8 poden ser utilitzats com els paràmetres de la funció. Els paràmetres superiors a 6 aniran a la pila.
- El registre R9 és utilitzat com a *Link Address* (LR).
- El registre R11 és utilitzat com el valor de retorn d'una funció (RV).
- El registre R12 és utilitzat com la part alta del valor de retorn de la funció per a implementacions de 32 bits (RVH).

A part dels GPR, també es defineix tot un segon conjunt de registres anomenats registres de propòsit especial (SPR). Aquests registres únicament poden ser accedits o modificats a través de les instruccions d'escriptura i lectura de registres especials, és a dir, les instruccions definides pel joc d'instruccions amb el mnemotècnic `l.mtspr` i `l.mfspr`.

Aquests registres especials estan dividits en grups segons les seves característiques, els principals estan recollits a la següent taula.

GRUP	DESCRIPCIÓ
0	Registres d'estat i de control (Sistema)
1	MMU de dades (DMMU)
2	MMU d'instruccions (IMMU)
3	Cache de dades (DC)
4	Cache d'instruccions (IC)
5	Unitat multiplicadora i acumuladora (MAC)

GRUP	DESCRIPCIÓ
6	Unitat de Debug (DU)
7	Unitat de Comptadors hardware (PCU)
8	Control de consum (PM, *power management*)
9	Controlador programable d'interrupcions (PIC)
10	*Tick Timer* (TT)

Taula 3-1. Grups principals dels registres de propòsit especial (SPR)

Qualsevol implementació que es faci necessita com a mínim que s'implementi els SPR del grup 0, el de sistema. Aquest grup conté registres de configuració per els comptadors hardware, les MMUs i les caches així com els registres VR (registre de versió), UPR (registre d'unitats presents), el CPUCFGR (el registre de configuració de la CPU), SR (registre de supervisió), el mapeig del PC (comptador de programa) i el mapeig dels GPR.

3.2.2 El joc d'instruccions

OR1K defineix 5 blocs d'instruccions simples i de 32 bits de mida uniforme distribuïts segon la seva categoria: ORBIS32, ORBIS64, ORPFX32, ORPFX64 i ORVDX64.

ORBIS32/64 defineix el joc d'instruccions bàsic d'OpenRISC, per a dades de 32 o 64 bits respectivament. Conté totes les instruccions habituals com les aritmètiques i lògiques, les d'accés a memòria, control del flux o instruccions especials. Per indicar que una instrucció pertany a un d'aquests dos grups, abans del mnemotècnic de la instrucció s'afegeix “l.”.

ORFPX32/64 és l'extensió per a les instruccions de coma flotant de simple precisió per a 32 bits o de doble precisió per a 64 bits. Poden ser utilitzades si la unitat de coma flotant (FPU) és present a la implementació. Per indicar que una instrucció pertany a un d'aquests dos grups, abans del mnemotècnic de la instrucció s'afegeix “lf.”.

Finalment, ORVDX64 és l'extensió per a les instruccions vectorials i DSP, operant amb dades de 8, 16, 32 i 64 bits. Per indicar que una instrucció pertany a aquest grup, abans del mnemotècnic de la instrucció s'afegeix “lv.”.

3.2.2.1 El format de les instruccions

Totes les instruccions OpenRISC tenen un codi d'operació de 6 bits que componen el codi màquina i els registres utilitzen 5 bits per a la seva identificació. Ara bé, segons la funcionalitat de cada instrucció, el format d'aquestes és diferent. Existeixen uns 7 formats diferents d'instruccions per a instruccions de:

- Codi operació + immediat (24 bits). Ex.: `l.j`
- Codi operació + registre destí, registre font A, registre font B + codi operacional per distingir instruccions entre el mateix Codi Op. (11 bits). Ex.: `l.add`
- Codi operació + registre destí, registre font A + reservat (8 bits) + codi operacional (2 bits) + immediat (6 bits). Ex.: `l.slli`
- Codi operació + registre destí, registre font A + immediat (16 bits). Ex.: `l.addi`
- Codi operació + primera part immediat (5 bits) + registre font A, registre font B + segona part immediat (11 bits). Ex.: `l.mtspr`
- Codi operació + codi operacional per distingir instruccions (5 bits), registre font A, registre font B + 11 bits reservats. Ex.: `l.sfeq`
- Codi operació + codi operacional per distingir instruccions (5 bits), registre font A + immediat (16 bits). Ex.: `l.sfeqi`

Existeixen algunes instruccions que no segueixen exactament aquests formats descrits, com per exemple la instrucció `l.movhi`, però solen mantenir una estructura organitzada.

El format concret de cada instrucció és el que es pot veure a la Taula 3-2. Per millorar la comprensió, s'ha seguit una notació específica:

- L'immediat s'expressa com a Imm, on cada bit de l'immediat es representa amb una “`n`”.
- Els guions “`-`” indiquen que el bit present és marcat com a reservat per a l'arquitectura.
- Rd, Ra i Rb indiquen registre destí, registre font A i registre font B respectivament.
- En una mateixa fila, els bits marcats com a “`f`” indiquen el codi operacional. Els mnemotècnics segueixen una llista ordenada des de 0 fins a 2 elevat el nombre de bits “`f`” - 1. Si el valor resultant no correspon a un mnemotècnic es marca amb “`-`”.
- A la part superior es pot seguir la numeració de bits.

Codi Op. (31–26)	Rd / Imm / Codi Op. (25–21)	Ra / Imm (20–16)	Rb / Imm (15–11)	Imm / Codi Op. / Reservat (10–0)	Mnemotècnic
0 0 0 0 0 0	n n n n n	n n n n n	n n n n n	n n n n n n n n n n n	l.j
0 0 0 0 0 1	n n n n n	n n n n n	n n n n n	n n n n n n n n n n n	l.jal
0 0 0 0 1 1	n n n n n	n n n n n	n n n n n	n n n n n n n n n n n	l.bnf
0 0 0 1 0 0	n n n n n	n n n n n	n n n n n	n n n n n n n n n n n	l.bf
0 0 0 1 0 1	0 1 - - -	- - - - - -	n n n n n	n n n n n n n n n n n	l.nop
0 0 0 1 1 0	Rd	- - - - 0	n n n n n	n n n n n n n n n n n	l.movhi
0 0 0 1 1 0	Rd	- - - - 1	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0	l.macrc
0 0 1 0 0 0	0 0 0 0 0	0 0 0 0 0	n n n n n	n n n n n n n n n n n	l.sys
0 0 1 0 0 0	0 1 0 0 0	0 0 0 0 0	n n n n n	n n n n n n n n n n n	l.trap
0 0 1 0 0 0	1 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0	l.msync
0 0 1 0 0 0	1 0 1 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0	l.psync
0 0 1 0 0 0	1 1 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0	l.csync
0 0 1 0 0 1	- - - - - -	- - - - - -	- - - - - -	- - - - - - - - - - - - -	l.rfe
0 0 1 0 1 0	*Reservat per a les instruccions ORVDX64 (lv.)*				
0 1 0 0 0 1	- - - - - -	- - - - - -	Rb	- - - - - - - - - - - - -	l.jr
0 1 0 0 1 0	- - - - - -	- - - - - -	Rb	- - - - - - - - - - - - -	l.jalr
0 1 0 0 1 1	- - - - - -	Ra	n n n n n	n n n n n n n n n n n	l.maci
0 1 1 1 0 0	- - - - - -	- - - - - -	- - - - - -	- - - - - - - - - - - - -	l.cust1
0 1 1 1 0 1	- - - - - -	- - - - - -	- - - - - -	- - - - - - - - - - - - -	l.cust2
0 1 1 1 1 0	- - - - - -	- - - - - -	- - - - - -	- - - - - - - - - - - - -	l.cust3
0 1 1 1 1 1	- - - - - -	- - - - - -	- - - - - -	- - - - - - - - - - - - -	l.cust4
1 0 0 0 0 0	*Reservat per a la instrucció ORBIS64 load (l.ld)*				
1 0 0 0 0 1	Rd	Ra	n n n n n	n n n n n n n n n n n	l.lwz
1 0 0 0 1 0	Rd	Ra	n n n n n	n n n n n n n n n n n	l.lws
1 0 0 0 1 1	Rd	Ra	n n n n n	n n n n n n n n n n n	l.lbz
1 0 0 1 0 0	Rd	Ra	n n n n n	n n n n n n n n n n n	l.lbs
1 0 0 1 0 1	Rd	Ra	n n n n n	n n n n n n n n n n n	l.lhz
1 0 0 1 1 0	Rd	Ra	n n n n n	n n n n n n n n n n n	l.lhs
1 0 0 1 1 1	Rd	Ra	n n n n n	n n n n n n n n n n n	l.addi
1 0 1 0 0 0	Rd	Ra	n n n n n	n n n n n n n n n n n	l.addic
1 0 1 0 0 1	Rd	Ra	n n n n n	n n n n n n n n n n n	l.andi
1 0 1 0 1 0	Rd	Ra	n n n n n	n n n n n n n n n n n	l.ori
1 0 1 0 1 1	Rd	Ra	n n n n n	n n n n n n n n n n n	l.xori
1 0 1 1 0 0	Rd	Ra	n n n n n	n n n n n n n n n n n	l.muli

Codi Op. (31–26)	Rd / Imm / Codi Op. (25–21)	Ra / Imm (20–16)	Rb / Imm (15–11)	Imm / Codi Op. / Reservat (10–0)	Mnemotècnic
1 0 1 1 0 1	Rd	Ra	n n n n n	n n n n n n n n n n n	l.mfspr
1 0 1 1 1 0	Rd	Ra	– – – – –	– – – f f n n n n n n	l.slli, l.srli, l.srai, l.rori
1 0 1 1 1 1	0 0 f f f	Ra	n n n n n	n n n n n n n n n n n	l.sfeqi, l.sfnei, l.sfgtui, l.sfgeui, l.sfltui, l.sfleui, –, –
	0 1 f f f				–, –, l.sfgtsi, l.sfgesi, l.sfltsi, l.sflesi, –, –
1 1 0 0 0 0	n n n n n	Ra	Rb	n n n n n n n n n n n	l.mtspr
1 1 0 0 0 1	– – – – –	Ra	Rb	– – – – – – – 0 0 0 1	l.mac
				– – – – – – – 0 0 1 0	l.msb
1 1 0 0 1 0	*Reservat per a les instruccions ORFPX32/64 (lf.)*				
1 1 0 1 0 0	*Reservat per a la instrucció ORBIS64 store (l.sd)*				
1 1 0 1 0 1	n n n n n	Ra	Rb	n n n n n n n n n n n	l.sw
1 1 0 1 1 0	n n n n n	Ra	Rb	n n n n n n n n n n n	l.sb
1 1 0 1 1 1	n n n n n	Ra	Rb	n n n n n n n n n n n	l.sh
1 1 1 0 0 0	Rd	Ra	– – – – –	– 0 0 f f – – 1 1 0 0	l.exths, l.extbs, l.exthz, l.extbz
			– – – – –	– 0 0 0 f – – 1 1 0 1	l.extws, l.extwz
			Rb	– 0 0 – – – – 0 f f f	l.add, l.addc, l.sub, l.and, l.or, l.xor, –, –
			Rb	– 0 0 – – – – 1 1 1 f	l.cmov, l.ff1
			Rb	– 0 0 f f – – 1 0 0 0	l.sll, l.srl, l.sra, l.ror
			Rb	– 0 1 – – – – 1 1 1 1	l.fl1
			Rb	– 1 1 – – – – 0 1 1 0	l.mul
			Rb	– 1 1 – – – – 1 0 f f	l.div, l.divu, l.mulu, –
1 1 1 0 0 1	0 0 f f f	Ra	Rb	– – – – – – – – – – –	l.sfeq, l.sfne, l.sfgtu, l.sfgeu, l.sfltu, l.sfleu, –, –
	0 1 f f f				–, –, l.sfgts, l.sfges, l.sflts, l.sfles, –, –
1 1 1 1 0 0	– – – – –	– – – – –	– – – – –	– – – – – – – – – – –	l.cust5
1 1 1 1 0 1	– – – – –	– – – – –	– – – – –	– – – – – – – – – – –	l.cust6
1 1 1 1 1 0	– – – – –	– – – – –	– – – – –	– – – – – – – – – – –	l.cust7
1 1 1 1 1 1	– – – – –	– – – – –	– – – – –	– – – – – – – – – – –	l.cust8

Taula 3-2. Codi màquina de les instruccions ORBIS32

3.3 ORPSoC

Amb l'especificació definida, el projecte OpenRISC ha continuat amb la implementació de processadors OpenRISC. N'existeixen 3 de principals: AltOr32, mor1kx i OR1200.

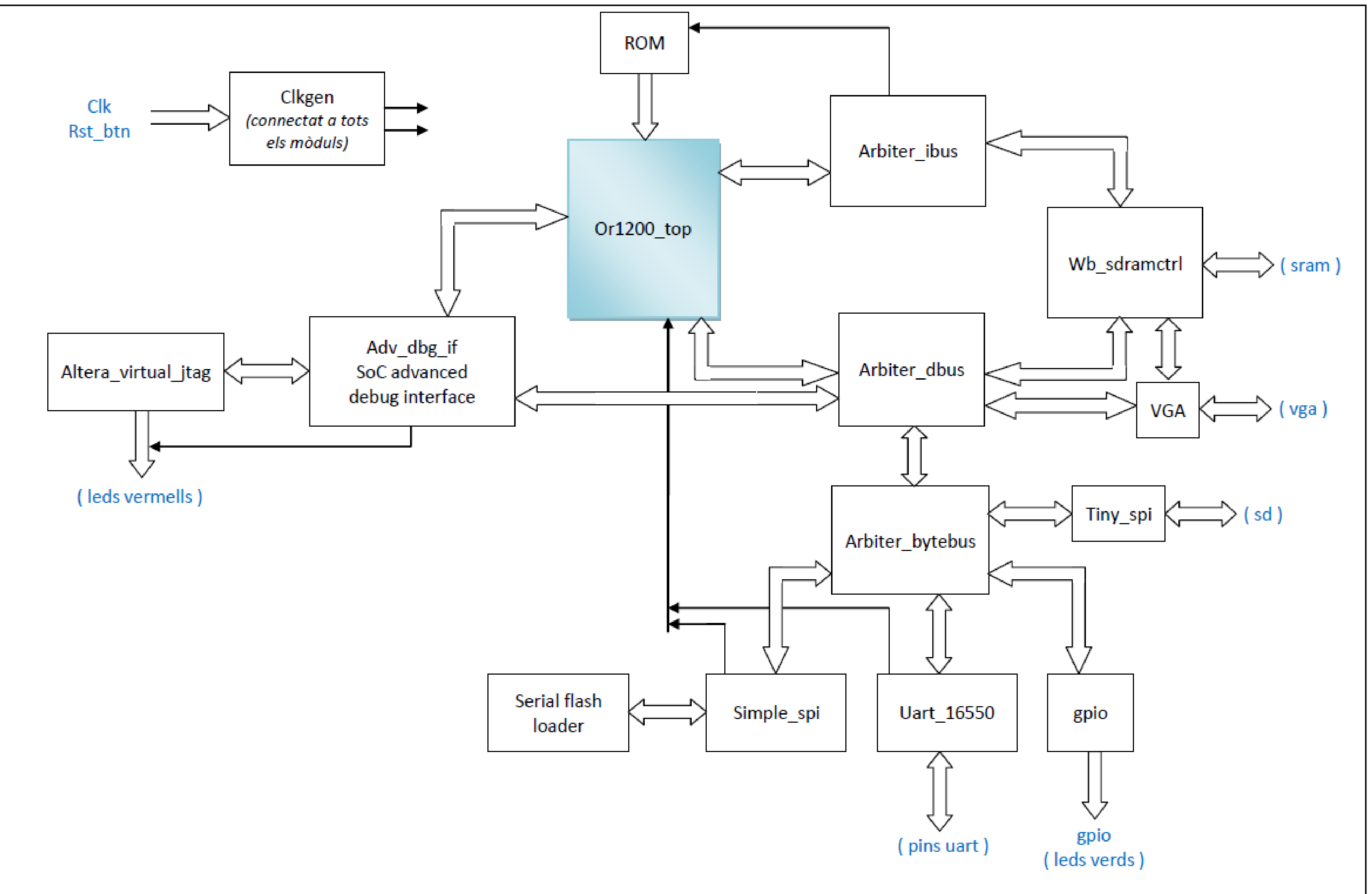
També s'han creat SoCs que contenen algun dels anteriors processadors i afegeixen tots els *cores* per tenir tot un sistema basat en OpenRISC però alhora interactuable, sense disposar tant sols d'un simple processador. Els dos SoCs disponibles són MinSoC i ORPSoC. A efectes pràctics tant és quin dels dos utilitzar, però s'ha escollit ORPSoC perquè disposa d'uns quants mòduls més i s'està treballant en una nova versió.

Així doncs, s'ha escollit ORPSoCv2, la segona versió d'ORPSoC, tot i que a mig projecte ha sorgit disponible la tercera versió. S'ha decidit no canviar de versió perquè la meitat de la feina ja està feta, la documentació s'ha fet amb la segona versió i tampoc hauria de suposar gaire feina actualitzar de versió amb la feina realitzada, però és innecessari per als objectius del projecte.

ORPSoC és el SoC de la plataforma de referència d'OpenRISC, de l'anglès OpenRISC *Reference Platform System on Chip.* Com a processador implementa l'OR1200, el processador basat òbviament en OpenRISC, al costat de *cores* com el controlador de pantalla VGA o el controlador UART. També disposa d'uns *scripts* de configuració, ports amb les diferents implementacions per a diferents plaques de desenvolupament, jocs de prova i la configuració per a la simulació, programació i execució d'un codi sobre aquest entorn.

A la Il·lustració 3-1 es pot veure els mòduls amb la configuració que actualment componen l'ORPSoCv2 per a la placa DE1 d'Altera, la placa física que s'ha utilitzat durant el projecte.

En concret, l'ORPSoC consta de més mòduls que poden ser activats o no segons la configuració final del processador que es desitgi. El que passa però, és que alguns d'aquests components no tenen un element físic per ser connectats en aquesta placa en concret, com per exemple el mòdul per a la connexió d'ethernet. És per això que a la il·lustració només es mostren els elements i mòduls disponibles a la placa a utilitzar.



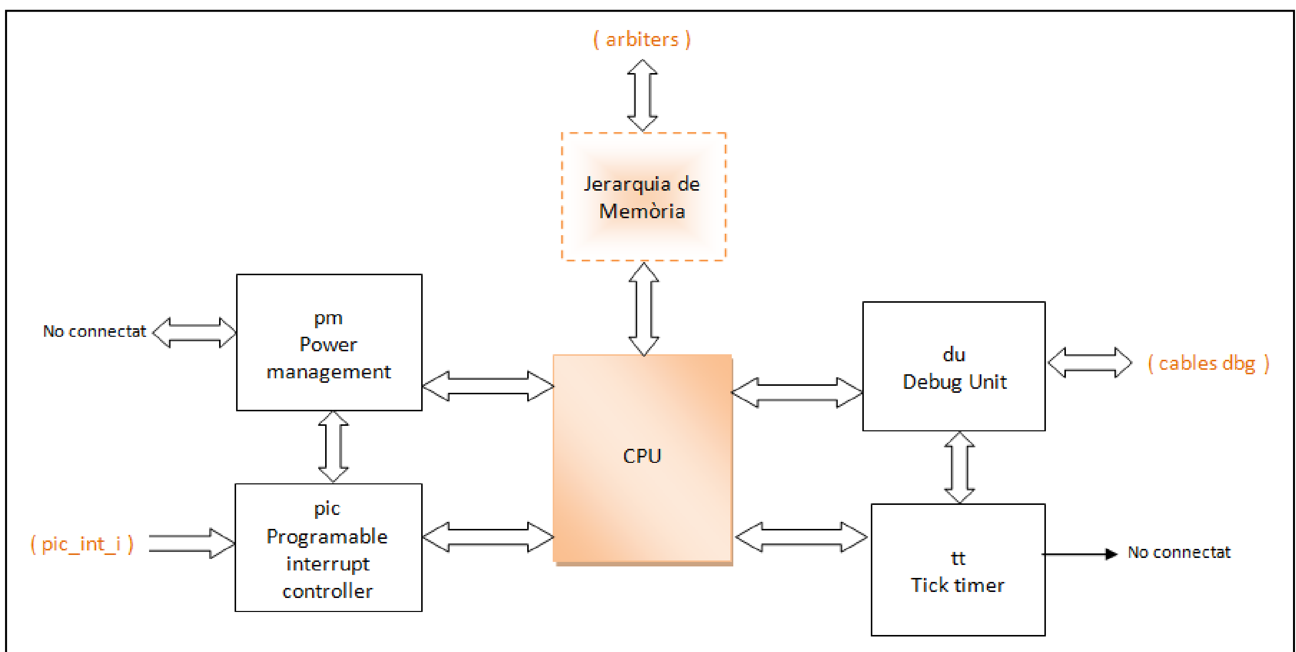
Il·lustració 3-1. Components ORPSoCv2 per a la placa DE1

Com es pot apreciar, des de fora cap al nucli, hi ha els elements físics de la placa connectats als controladors, que al seu pas, acaben interconnectats a “Or1200_top”, que és el processador OpenRISC implementat.

3.4 El processador OR1200

El processador OR1200 és un processador basat en OpenRISC que s’ha implementat seguint l’especificació OR1K.

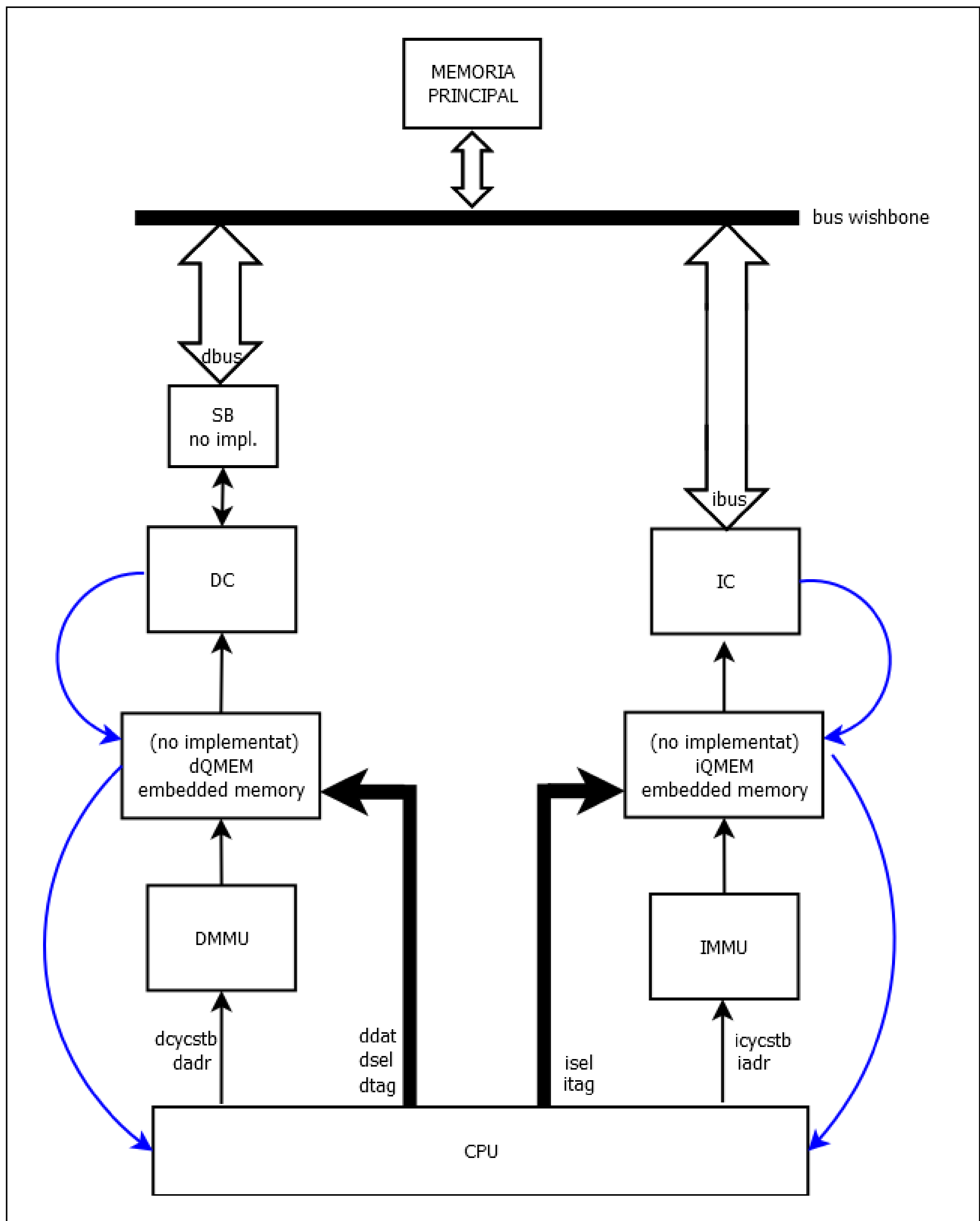
El processador està descrit amb llenguatge HDL Verilog composat en mòduls. Com es pot veure a la Il·lustració 3-2, la CPU està connectada a la jerarquia de memòria i als mòduls complementaris com la unitat de debug o el controlador d’interrupcions. Igualment com passava al cas anterior, el processador té un fitxer completament configurable per activar i desactivar mòduls i canviar característiques. Així doncs, a la il·lustració es mostren els mòduls actualment implementats, encara que n’hi ha que no estiguin connectats enlloc, perquè estan desactivats per a la configuració de la placa física DE1 que s’ha utilitzat.



Il·lustració 3-2. Processador OR1200: mòdul “or1200_top”

El processador utilitza una arquitectura Harvard, separant el camí de memòria de dades i d’instruccions. En concret, l’accés a memòria és bastant complet constant del suport de memòria virtual (MMU) i de caches, a part de la memòria principal, com es pot veure a la

Il·lustració 3-3 següent. Les senyals i busos representats sortint de la CPU són anomenats amb la nomenclatura utilitzada al propi processador.



Il·lustració 3-3. Jerarquia de memòria OR1200

El mòdul QMEM consisteix en una petita memòria integrada dins l'àrea que ocupa el processador. La raó principal per disposar d'aquesta memòria és poder posar-hi funcions crítiques i obtenir un accés ràpid. El problema però, és que ocupa més espai i fa anar tot el sistema més lent (les cache i la memòria principal estan a un nivell superior després d'aquesta memòria), així que per defecte ve desactivada i, per tant, quan es genera el projecte, el mòdul no s'implementa, simplement queden connectades les entrades amb les sortides corresponents.

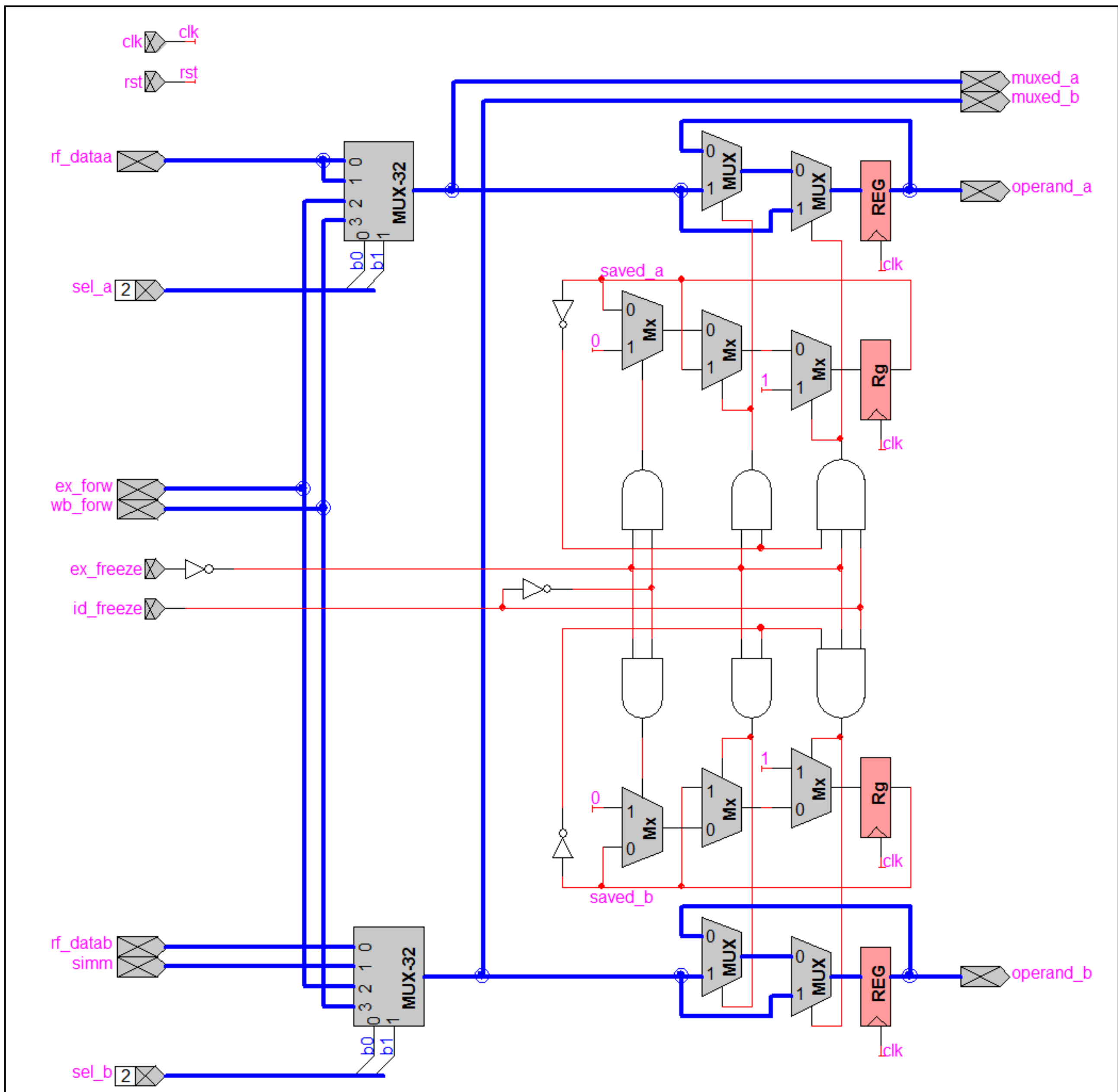
El camí de dades també disposa d'un buffer d'escriptures però també incrementa la mida final del disseny i ve desactivat per defecte. Així doncs, el camí actual passa de la CPU a les MMUs, de les MMUs a les caches i a traves del bus *Wishbone* cap a memòria principal.

3.4.1 El pipeline

El processador OR1200 està segmentat en etapes però observant el codi font no es detecten fàcilment les etapes amb què està segmentat ni quina és la quantitat d'etapes de segmentació.

Per poder determinar les etapes, doncs, s'ha localitzat tots els registres que travessen les senyals i busos, extraient-los de tots els fitxers separats per mòduls. El principal problema que ha sorgit en aquesta part ha estat precisament això, passar d'un codi organitzat en mòduls a una organització en etapes. Per fer-ho d'una manera fàcil i entenedora, i alhora que servís per comprendre com funciona tot el processador, s'ha anat dibuixant la circuiteria mòdul a mòdul, posant al mateix nivell els registres que es corresponien a la mateixa etapa. Un cop fet tot això, s'ha anat ajuntat tots els dibuixos per acabar d'organitzar els registres en etapes i finalment s'ha generat un esquema del pipeline real del processador.

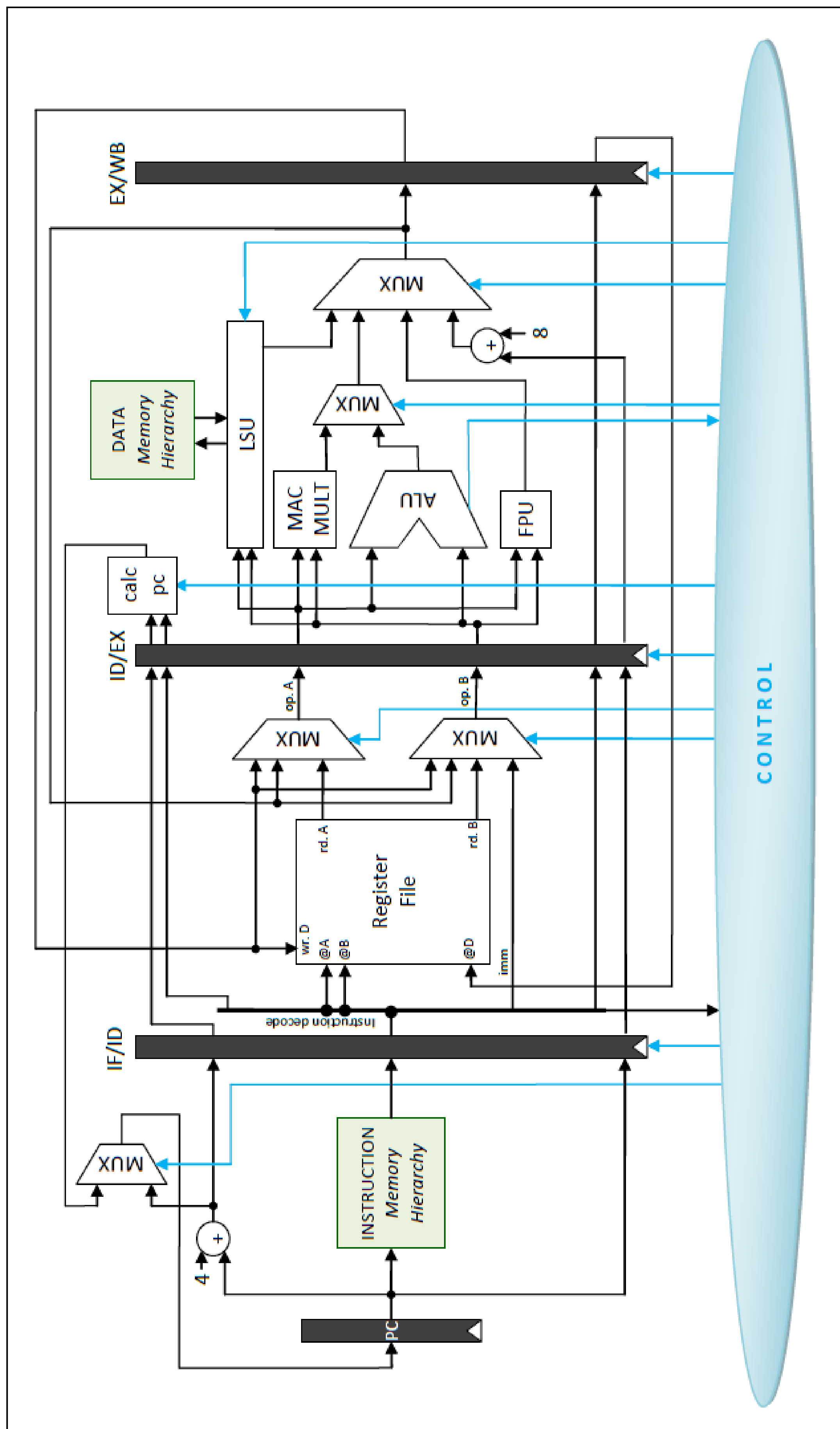
La Il·lustració 3-4 mostra, com a exemple, un dels fitxers convertit a circuit pel mitjà que s'ha descrit per fer palesa la feina realitzada. A l'Annex C es pot trobar tot el conjunt de circuits a partir del codi que s'han realitzat.



Il·lustració 3-4. Circuit del fitxer or1200_operandmuxes.v

Aquest fitxer descriu els multiplexors per inserir a l'etapa d'execució les dades, segons sigui un immediat, provingui del banc de registres o provingui dels curtcircuits d'etapes posteriors. Clarament, es pot determinar que `operand_a` i `operand_b` ja pertanyen a l'etapa d'execució, perquè les `rf_dataa` i `rf_datab` són les dades llegides del banc de registres (en etapa de lectura), i `simm` prové de l'etapa de descodificació del mòdul que controla el processador. Per tant, es pot determinar que els registres són els que separen l'etapa de descodificació amb la d'execució.

Així doncs, identificats els registres, s'ha pogut obtenir el diagrama del pipeline recollit a la Il·lustració 3-5. S'hi pot identificar tot el camí de dades i les senyals provinents de control.



Il·lustració 3-5. Pipeline del processador OR1200

Veient aquesta il·lustració es pot determinar que el pipeline obtingut està dividit en 4 etapes. Tal com es descriu en la documentació del processador OR1200 [15], el pipeline està segmentat en 5 etapes, les etapes: Fetch (IF), descodificació (ID), execució (EX), memòria (MEM) i escriptura (WB).

Això podria semblar una incongruència, però és correcte. El pipeline del processador en realitat està segmentat en 4 o 5 etapes. Sempre hi ha 4 etapes regulars (IF, ID, EX, WB) que són les que s'han identificat a la il·lustració i la majoria d'instruccions segueixen aquesta segmentació. Però a més a més, sempre que hi ha un accés a memòria (sigui d'un load o d'un store), una etapa més és afegida (l'etapa de MEM), obtenint llavors un pipeline amb les etapes IF, ID, EX, MEM i WB. Això s'aconsegueix bloquejant el pipeline de les altres etapes, accedint a memòria i prosseguint l'execució normal altra vegada quan l'accés a memòria ja es troba a l'etapa WB.

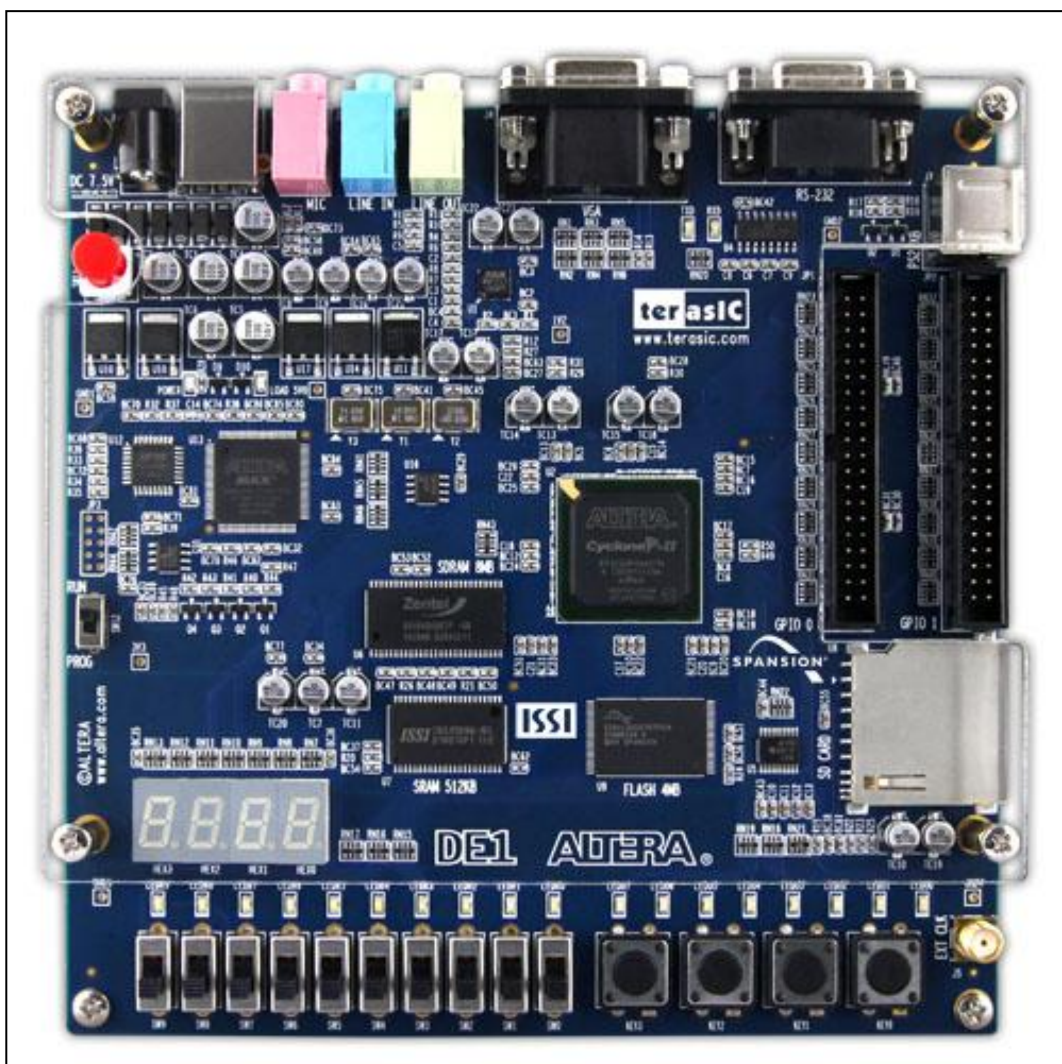
Aquesta situació s'ha descobert mentre s'analitzava el codi font per obtenir els circuits, i també ha servit per corroborar la informació procedent de la pàgina principal del processador.

4 EINES I RECURSOS SW I HW

En aquest capítol es parlarà de les eines i dels recursos software i hardware que són necessaris per a la realització del projecte.

4.1 Placa DE1 d'Altera

La placa DE1 (*Development & Education board*) de Terasic per a Altera, Il·lustració 4-1, és una placa amb tot un conjunt de components que estan connectats directament a la FPGA (xip més gran) perquè pugui ser configurada per implementar qualsevol tipus de sistema amb la disponibilitat d'aquests components.



Il·lustració 4-1. Placa DE1 d'Altera

Els components de la placa són:

- FPGA Altera Cyclone II EP2C20F484C7N amb 18.752 LEs
- Dispositiu de configuració sèrie d'Altera (EPCS4) per a Cyclone II 2C20
- Suport de programació per JTAG i AS (*Active Serial*)
- SDRAM de 8Mbyte (1M x 4 x 16)
- Memòria flash de 4Mbyte
- SRAM de 512Kbyte(256Kx16)
- Ranura per a targeta SD
- Polsadors
- 10 selectors DPDT
- 8 LEDs verds d'usuari
- 10 LEDs vermells d'usuari
- 4 Visualitzadors LED de 7 segments
- Oscil·lador de 50MHz, de 24MHz, de 27MHz i rellotge extern
- CODEC d'àudio de 24 bits amb jacks de línia d'entrada, de sortida i de micròfon
- Connector VGA amb convertidor digital analògic (DAC) VGA 4 bits R-2R per canal
- Transceptor RS-232 i connector de 9 pins
- Connector PS/2 per a teclat o ratolí
- 2 capçals d'expansió de 40 pins

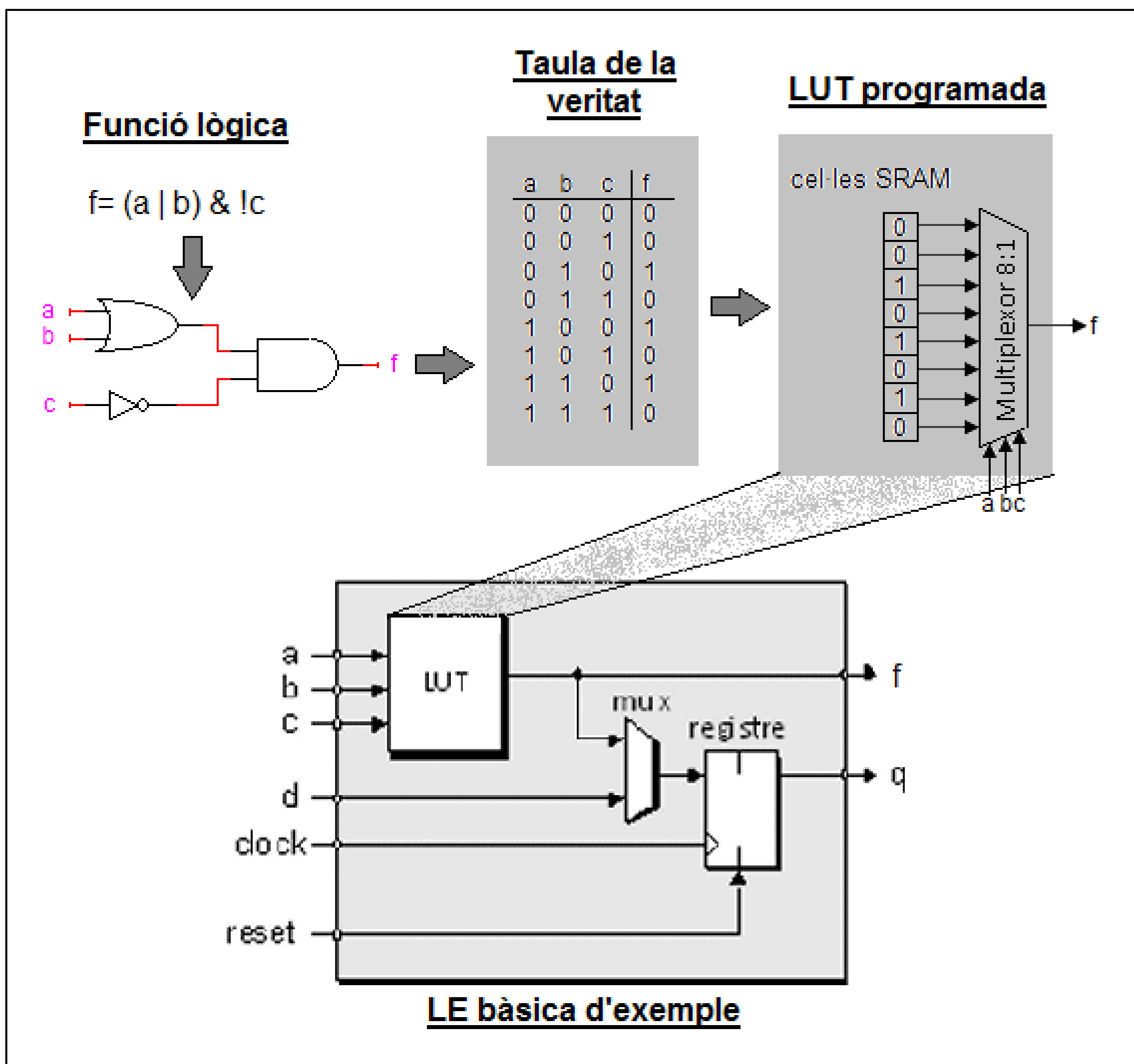
Aquesta placa és l'entorn físic on s'ha provat el processador OpenRISC. En concret es programa la FPGA amb el processador i es configuren algun dels components per utilitzar-los, com els LEDs vermells i verds, però també es poden configurar altres components com el VGA, la targeta SD i els polsadors o selectors.

4.1.1 Les FPGA

Una FPGA, de l'anglès *Field Programmable Gate Array* [16], és un dispositiu que apareix inicialment com a convergència entre els PLD i els ASIC, degut a la diferència que hi havia entre els dos. Per una banda, hi ha els dispositius lògics programables (PLD) que són molt senzills però molt configurables, i en canvi, per l'altra, hi ha els circuits integrats d'aplicació específica (ASIC) que són cars i costosos de dissenyar, però suporten funcions molt complexes.

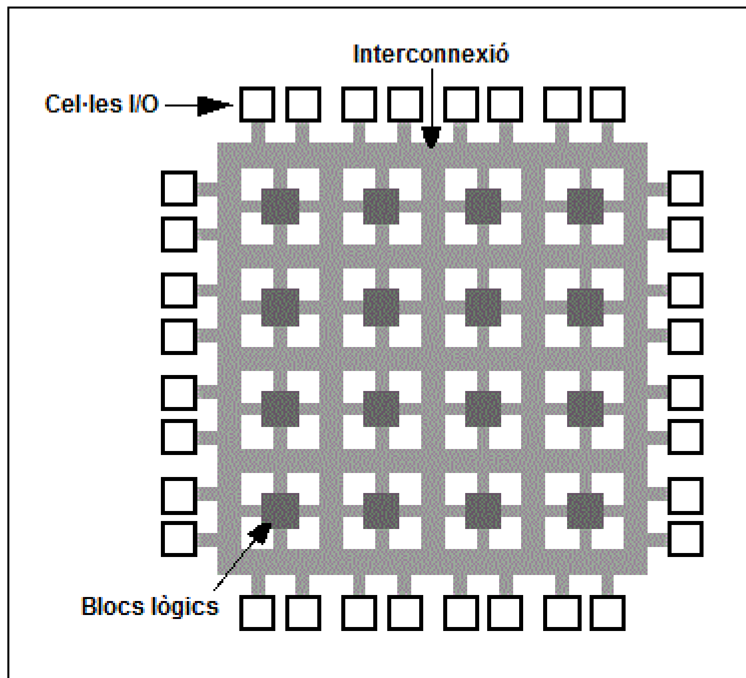
Les FPGA són bastant semblants a un ASIC, perquè poden contenir funcions molt complexes, però es diferencien en el fet que són programables, és a dir, són uns circuits integrats tals que la seva funcionalitat lògica es defineix posteriorment a la seva construcció i fabricació. Així doncs, les FPGA contenen molts blocs de lògica configurables i programables amb connexions a aquests blocs també configurables.

Aquests blocs de lògica configurables s'anomenen LE (*Logic Elements*) i no estan formats per portes lògiques (com s'acaba de dir) sinó que es compon bàsicament d'una LUT, un multiplexor i un registre (com a una implementació bàsica). Una LUT (*Lookup Table*) és una taula de cerca on depenent de les entrades, se selecciona directament el resultat de la funció a implementar, i per tant, no existeix físicament el disseny lògic. La Il·lustració 4-2 mostra per a una funció lògica què és el que s'emmagatzema.



Il·lustració 4-2. Configuració d'una LUT d'exemple

Així mateix, al voltant de cada LE hi ha la xarxa d'interconnexió entre els LE i els blocs d'entrada i sortida, formant una xarxa matricial. L'esquema de la Il·lustració 4-3 mostra la composició bàsica d'aquesta interconnexió.



Il·lustració 4-3. Esquema de la interconnexió d'una FPGA

Per programar una FPGA es necessita un codi font descrit amb un llenguatge de descripció de hardware (HDL). Per passar del codi font a una implementació dins de la FPGA es realitza la compilació que consisteix dels diferents processos següents:

- **Síntesi**: El codi es sintetitza en un circuit format pels elements lògics disponibles a la FPGA.
- ***Fitting***: És el procés de muntatge, dividit en dues tasques:
  - ***Placement***: Consisteix en determinar la ubicació dels LE que s'han definit a la FPGA.
  - ***Routing***: Consisteix en escollir les rutes de connexió (encaminament) per connectar els diferents LE que s'han programat.

Amb tot això, s'obté finalment un fitxer binari anomenat *bitstream*, que pot ser descarregat directament a la FPGA.

4.2 Software Quartus II d'Altera

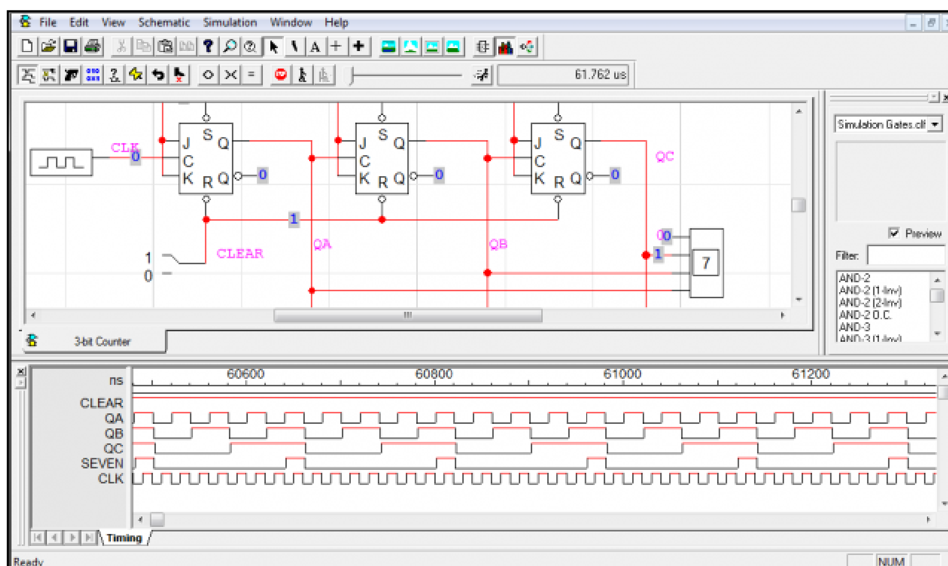
La FPGA de la placa DE1, la Cyclone II EP2C20F484C7N, pertany al fabricant Altera [17]. El mateix fabricant proporciona un software per tal que pugui ser programada.

El programari Quartus II ofereix totes les eines necessàries, com poden ser l'editor HDL, el compilador i programador a la FPGA, les eines de generació de circuits o eines d'anàlisi temporal i de consum.

Quartus II serà qui generi el *bitstream* del codi del processador i qui el programi dins la FPGA, tasca essencial per utilitzar la placa. En concret, s'ha utilitzat la versió Quartus II Web Edition, que és la versió amb llicència gratuïta. Aquesta versió disposa de menys característiques però no és cap limitació pel projecte que es realitza.

4.3 Software LogicWorks

LogicWorks és un dissenyador de circuits lògics interactiu [18]. Permet crear els mòduls desitjats, connectar tot un circuit i disposa d'un simulador de senyals per simular tot el disseny realitzat.



Il·lustració 4-4. Software LogicWorks

Amb aquesta eina s'han dibuixat els circuits del codi OpenRISC de l'Annex C, ja que és una eina fàcil d'utilitzar i molt intuïtiva.

4.4 Simulador ModelSIM

Tot codi necessita ser provat abans d'executar-lo físicament perquè si s'ha produït algun error i es prova directament a la placa física, es poden malmetre els components. Per això, una manera de testejar el codi és simular-lo en un simulador que es comporti de la mateixa forma com si del hardware mateix es tractés.

En un principi, s'havia previst utilitzar el simulador ModelSIM-Altera Starter Edition, la versió gratuïta del simulador ModelSIM de Mentor Graphics amb integració per Altera. El problema ha estat que aquesta versió gratuïta presenta unes limitacions més severes (redueix excessivament la velocitat de la simulació pel fet de disposar d'un codi font gran) i al simular el codi, els resultats no arribaven fins al cap de 5 minuts per a una execució de 30 segons.

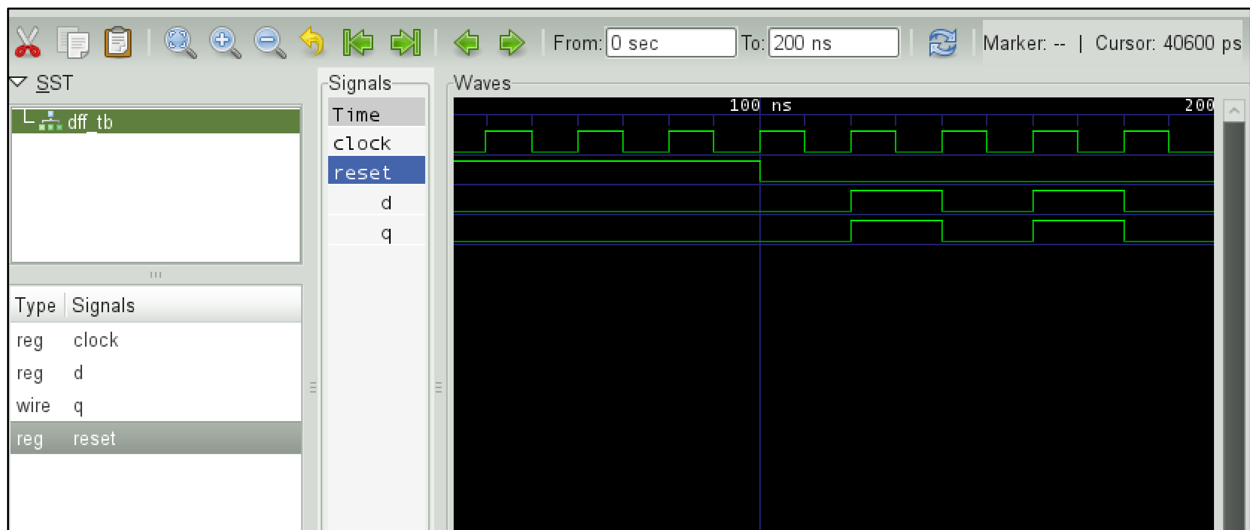
És necessari utilitzar aquest simulador si es vol simular el processador amb la configuració tal com aniria dins la placa física, ja que al ser una integració amb Altera, té els recursos necessaris que defineix únicament Altera com funcions específiques d'aquest fabricant.

Com que la simulació és lenta, s'ha decidit buscar una alternativa procedint amb el següent criteri: Com que els canvis de codi es fan sobre el processador base i no sobre la versió específica de la placa, no cal que s'utilitzi un simulador específic per a la placa per fer proves. No obstant, un cop determinat que funciona correctament, s'utilitzarà una simulació lenta del ModelSIM per comprovar que tot funciona sobre la placa simulada. Així doncs, s'aconseguirà la velocitat desitjada, però també es provarà que funcioni abans no es programi físicament a la placa, com a pas final.

4.5 Simulador Icarus Verilog + Visualitzador GTKWave

Com alternativa, s'ha utilitzat el simulador de codi lliure Icarus Verilog [19]. El simulador ofereix el valor i l'estat de totes les senyals involucrades en tot moment, segons un marcatge temporal. El resultat de tota l'execució es pot exportar en un fitxer de bolcatge VCD (*value change dump*), per a una posterior visualització.

Per poder visualitzar el resultat, s'ha utilitzat el software de codi lliure GTKWave [20] que es tracta d'un visualitzador de senyals. Al visualitzador se li ha entrat el fitxer VCD obtingut prèviament amb el simulador Icarus.



Il·lustració 4-5. Visualitzador de senyals GTKWave

4.6 Tool chain d'OpenRISC

La plataforma OpenRISC es completa amb un conjunt d'eines útils agrupat en la OpenRISC GNU *tool chain* [21] descarregables des del seu web. La *tool chain* conté eines basades en projectes de GNU, que s'han portat a l'arquitectura OpenRISC. Les eines destacables, de les quals alguna s'ha utilitzat es descriuen a continuació.

Existeix una versió del compilador GCC per tal que pugui compilar codis i aquests siguin executables pels diferents processadors d'OpenRISC. És a dir, crea els fitxers binaris que segueixen l'arquitectura OR1k i pugin ser executables, per exemple, pel processador OR1200. També disposa d'altres eines pertinents com l'assemblador. El compilador serà útil per generar els jocs de prova per testejar el processador OR1200.

Existeix també el depurador GDB (GDB *debugger*) en la seva versió per OpenRISC. Pot ser útil per depurar i analitzar els codis a través de la unitat de debug del propi processador. No s'ha utilitzat perquè s'han obtingut els cronogrames de les senyals on s'ha comprovat que els valors ja eren correctes.

4.6.1 Simulador OR1ksim

OpenRISC també disposa d'un simulador per a la seva arquitectura. L'OR1ksim és capaç d'emular sistemes basats en OpenRISC al nivell d'instruccions, tal com diu la seva descripció [22].

La idea bàsica és la de poder crear aplicacions per a un processador OpenRISC (per exemple, disposar de codis executables per a OpenRISC) i poder-los simular en aquest simulador que és configurable, per definir quin tipus d'implementació es desitja: quin tipus de processador OpenRISC es vol utilitzar, quins dispositius perifèrics s'han d'activar, les mides i configuració de memòria, etc.

El propi simulador es comporta com un processador i per tant no s'ha utilitzat. El que s'ha fet és simular la implementació del processador, concepte diferent.

4.6.2 Imatge de Linux

Finalment, també han generat una imatge de Linux funcional sobre un entorn basat en OpenRISC. Concretament, utilitzant el depurador i el simulador OR1ksim ja es podria observar el seu funcionament sense disposar d'una implementació física, carregant-hi aquesta imatge.

4.7 Or1k-tcltools

L'or1k-tcltools és una sèrie d'*scripts* Tcl creats per Franck Jullien, un membre col·laborador del projecte OpenRISC [23]. En concret, aquests *scripts* serveixen per carregar a memòria un binari perquè aquest pugui ser executat pel processador OpenRISC.

Per utilitzar-lo és necessari programar primerament l'FPGA amb el processador OpenRISC per comunicar l' "*Advanced debug interface*" del processador amb l'Altera Virtual JTAG. A través d'aquesta comunicació, es carrega a memòria el binari compilat per OpenRISC.

L’ork-tcltools és una solució específica per a plataformes d’Altera, i per tant, si es disposa de plaques d’altres fabricants s’han de buscar altres alternatives no contemplades en aquest projecte.

4.8 Eines de l’ORPSoC

Al capítol 3, quan s’ha introduït l’ORPSoC, s’ha comentat que disposava d’*scripts* de configuració i jocs de prova. En aquest apartat es detallaran els directoris importants presents al SoC i les comandes més útils per tal d’interactuar amb el processador.

Els directoris amb cert interès són:

- `/boards`: Aquest directori conté les diferents implementacions específiques per a plaques i FPGA de diferents models i fabricants. En concret, dins de `/boards/altera/de1/` hi ha els mòduls que difereixen d’una implementació genèrica i el necessari per sintetitzar el projecte a la placa utilitzada.
- `/rtl`: Aquest directori conté el codi font global del processador. Les variacions específiques que afecten a la placa amb la que s’ha treballat es troben al subdirectori corresponent del directori `/boards`.
- `/sim`: Aquest directori permet simular el codi RTL del processador general. Consta de 3 subdirectoris: `run/` per començar la simulació, `out/` per emmagatzemar els resultats i `bin/` que és on hi ha les comandes a executar quan s’inicia la simulació amb `/run`.
- `/sw/tests`: És el directori on hi ha els jocs de prova del processador i de mòduls específics.

Al llarg de tot l’arbre de directoris del SoC, existeixen uns fitxers *Makefile* que ajuden a simplificar la feina, ja que per aconseguir amb èxit una sola tasca calen més d’una línia de comandes executades per terminal (en un sistema operatiu Linux). Segons el que es vulgui realitzar, en les següents subseccions es detallen les comandes d’utilitat.

4.8.1 Obtenir el fitxer binari per a l'FPGA

En aquest cas, es vol compilar el codi de la placa DE1 i obtenir el fitxer ".sof" que és el binari per programar la FPGA. Per això cal seguir els passos següents:

1. Obrir una terminal
2. Entrar al directori `/boards/altera/de1/syn/quartus/run`
3. És obligatori definir una variable amb la ruta a on estigui instal·lat el programa Quartus II d'Altera: `export ALTERA_PATH=<ruta software altera>`
4. Executar: `make all`

4.8.2 Programar el processador a l'FPGA

La FPGA es programa a la placa a través del cable USB per JTAG. Com que un cop programat el processador a la placa es voldrà que faci alguna cosa, o sigui, executi un codi, cal carregar també el binari del codi que hagi d'executar el processador a memòria.

Per programar el processador cal disposar del fitxer ".sof". Per això, caldrà prèviament seguir els passos de l'apartat anterior.

Programar el processador OR1200 dins l'FPGA:

1. Seguir els passos de l'apartat anterior per obtenir el fitxer ".sof" del processador
2. Entrar al directori `/boards/altera/de1/syn/quartus/run`
3. Executar: `make pgm`

Ara el processador ja ha estat programat dins l'FPGA. Per posar el binari OpenRISC d'un codi de prova a memòria perquè el processador executi quelcom, cal:

1. Disposar del codi original. Una forma fàcil de crear un codi fàcil és programar-lo en llenguatge assemblador, seguint el joc d'instruccions de l'arquitectura OpenRISC.
2. Compilar el codi. Un cop instal·lada la *tool chain*, disposarem del compilador, per tant: `or32-elf-as –o <nom_binari>.elf <nom_codi_prova>.S` (o amb `or32-elf-gcc`)
3. Disposant també dels *scripts* or1k-tcltools: `or1k-download <nom_binari>.elf`

Amb tot això, el processador OR1200 estarà programat dins l'FPGA i s'haurà executat el codi OpenRISC que acabem de carregar a memòria.

4.8.3 Simular el processador

Per simular el processador OR1200 sense necessitat de testejar-lo directament a la placa, els passos a seguir són els descrits a continuació. Cal destacar que s'expliquen els passos utilitzant el software descrit en aquest capítol, per a simular el processador genèric:

1. Entrar al directori `/sim/run`
2. Executar: `make rtl-tests TEST=<nom_test> VCD=1`
3. Executar: `make sim-gtkwave`

El nom del test al punt 2, si es vol testejar el processador, consisteix en el nom del fitxer sense l'extensió que es pot trobar al directori `/sw/tests/or1200/sim/`, com per exemple "or1200-basic". Per a la realització d'aquest projecte, s'ha creat un directori nou a `/sw/tests/cust/sim` per tal d'afegir-hi a allà els codis personalitzats que es vulguin crear.

L'opció VCD=1 habilita la generació del fitxer ".vcd" per la posterior visualització de resultats. Amb aquesta regla ja es crida al simulador Icarus Verilog i es fa el volcatge de les senyals.

La regla "sim-gtkwave" del punt 3 també s'ha afegit per a aquest projecte, la qual inicialitza el visualitzador GTKWave amb el fitxer ".vcd" generat a la última simulació. A més a més, també es carrega un fitxer de configuració "display.gtkw" ubicat a `/sim/bin/` creat perquè directament a l'obrir el visualitzador, el fitxer ".vcd" es mostri a la finestra temporal amb les senyals habitualment comunes, com pot ser la senyal de rellotge, la de reset, les dades llegides del banc de registres, el resultat de la ALU, etc.

No obstant, si en comptes de voler simular el processador general es vol simular el SoC que anirà directament a la placa, llavors s'han de seguir els mateixos passos però des de `/boards/altera/de1/sim/run`. En aquest cas però, s'ha d'afegir PRELOAD_RAM=1 al pas 2 per tal de carregar el programa a memòria. La regla del pas 3 no s'ha proveït perquè es pot visualitzar des del mateix ModelSIM i no cal utilitzar el visualitzador.

5 DISSENY I IMPLEMENTACIÓ DELS COMPTADORS HARDWARE

El manual de l'arquitectura defineix les característiques bàsiques del mòdul de comptadors hardware anomenat PCU (*Performance Counters Unit*). D'entrada deixa clar que la seva implementació és opcional, fet pel qual el mòdul PCU no estava implementat en les versions actuals del processador.

L'arquitectura defineix 8 comptadors obligatoris i especifica que altres comptadors addicionals poden ser definits per la pròpia implementació de la unitat. Aquests comptadors formen part del conjunt de registres de propòsit especial del processador.

5.1 Esdeveniments

Esdeve-niment	NOM (en anglès)	DESCRIPCIÓ
LA	Load Access event	Accessos per instruccions Load a MEM
SA	Store Access event	Accessos per instruccions Store a MEM
IF	Instruction Fetch event	Instruccions buscades (*fetch*)
DCM	Data Cache Miss event	Missos de Cache de dades
ICM	Instruction Cache Miss event	Missos de Cache d'instruccions
IFS	Instruction Fetch Stall event	Aturades d'IF
LSUS	LSU Stall event	Aturades de LSU
BS	Branch Stall event	Aturades per salts
DTLBM	Data TLB Miss event	Missos de la TLB de dades (MMU)
ITLBM	Instruction TLB Miss event	Missos de la TLB d'instruccions (MMU)
DDS	Data Dependency Stall	Aturades per dependència de dades
WPE	Watchpoint Events	Comptatge de Watchpoints (unitat de debug)

Taula 5-1. Esdeveniments definits pel manual d'arquitectura OR1K

Els esdeveniments definits per l'arquitectura són els presents a la Taula 5-1. Cal destacar que tota la descripció que el manual ofereix respecte a aquests esdeveniments, el que compten i com ho compten, és el que es mostra a la taula. No hi ha més informació i per tant, s'ha hagut de prendre decisions sobre el seu disseny per determinar exactament què compten en esdeveniments on el nom és ambigu, com ho fan i en quin moment compten.

5.1.1 Definició dels esdeveniments

LOAD ACCESS (LA)

Es considera que hi ha hagut un accés de load cada cop que es realitzi un accés de lectura a la jerarquia de memòria, corresponent a una instrucció load.

STORE ACCESS (SA)

Es considera que hi ha hagut un accés de store cada cop que es realitzi un accés d'escriptura a la jerarquia de memòria, corresponent a una instrucció store.

INSTRUCTION FETCH (IF)

Es considera que succeeix cada cop que es porti una instrucció de memòria per obtenir a la CPU la instrucció com a tal.

DATA CACHE MISS (DCM)

Sempre que la cache de dades estigui activada, es considera cada cop que falli per no estar carregada la línia de dades a la cache i, conseqüentment, s'hagi d'anar a memòria principal.

INSTRUCTION CACHE MISS (ICM)

Sempre que la cache d'instruccions estigui activada, es considera cada cop que falli per no estar carregada la línia a la cache i, conseqüentment, s'hagi d'anar a memòria principal.

IF STALL (IFS)

Es considera una aturada d'IF cada cop que es bloquegi el pipeline perquè s'està esperant que la següent instrucció arribi de memòria (faci el fetch). Aquest esdeveniment es compta en cicles.

LSU STALL (LSUS)

Aquest esdeveniment es considera com les aturades de la Unitat de Load/Store. En aquest sentit, succeeix a partir que una instrucció de load o store entra en etapa d'execució (EX) fins que arriba la confirmació de memòria. A diferència de l'esdeveniment anterior, no es considera com que el pipeline està aturat sinó que la LSU està ocupada, repercutint en que els dos esdeveniments puguin ocasionar-se al mateix instant. Aquest esdeveniment es compta en cicles.

BRANCH STALL (BS)

Es considera una atura degut a salts quan el pipeline estigui aturat pel fet d'assignar el nou PC, abans no es pugui fer fetch de la següent instrucció. Aquest esdeveniment es compta en cicles. Això significa que només afecten els cicles degut a salts, i no els d'IFS.

DATA TLB MISS (DTLBM)

Sempre que la MMU de dades estigui activada, es considera cada cop que el *tag* falli o no sigui vàlid al TLB.

INSTRUCTION TLB MISS (ITLBM)

Sempre que la MMU d'instruccions estigui activada, es considera cada cop que el *tag* falli o no sigui vàlid al TLB.

DATA DEPENDENCY STALL (DDS)

Es considera sempre que hi hagi aturades degut a riscos de dades. Aquest esdeveniment però, no es contemplarà degut a la implementació actual del processador. L'OR1200 és conservador, i independentment de si existeix riscos de dades o no, atura el pipeline sempre que no es garanteixi una execució de les 4 etapes bàsiques, com un accés a memòria, una divisió (que triga 32 cicles), etc. A més a més, es garanteix l'ordre del programa i disposa de curtcircuits de les etapes EX i WB a l'etapa de descodificació per tal que instruccions joves puguin tenir les dades del mateix registre d'instruccions velles que encara no han realitzat l'escriptura. Amb tot això, els riscos de dades queden resolts sense provocar aturades al pipeline.

WATCHPOINT EVENTS (WPE)

Existeixen 11 watchpoints a la unitat de debug. Sempre que la unitat de debug estigui present, aquest esdeveniment serveix per detectar quan passa un determinat watchpoint i

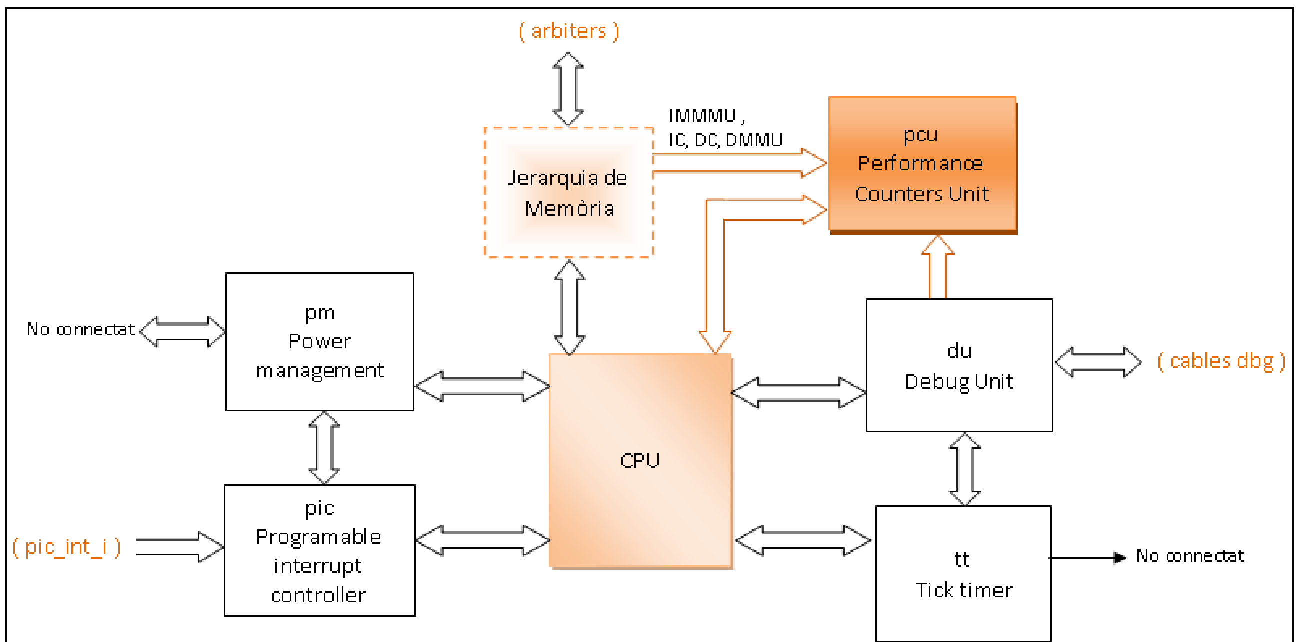
comptar-ne totes les ocurrències. Existeix un esdeveniment per cada un dels 11 watchpoints significant que es poden comptar totes les combinacions possibles activant i desactivant l'esdeveniment per a cada watchpoint.

5.2 Ubicació del mòdul PCU

Sabent quins són els esdeveniments que poden succeir i coneixent la seva descripció, podem saber quines són les unitats que produeixen aquests esdeveniments. En concret, són la cache de dades (DCM), la cache d'instruccions (ICM), la MMU de dades (DTLBM), la MMU d'instruccions (ITLBM), la unitat de debug (WPE) i finalment, la CPU (LA, SA, IF, IFS, LSUS i BS). Igualment, sabent també que formen part del conjunt de registres especials, caldrà proveir la interfície d'accés als registres especials, que procedeix també de la CPU.

El disseny d'aquest processador es caracteritza per ser d'un disseny modular, on cada mòdul nou correspon a un nou fitxer (o fitxers) i la desactivació d'un mòdul en concret no afecta al correcte funcionament del processador. En aquest sentit doncs, només cal proveir la interfície de connexió entre els mòduls, és a dir, les senyals necessàries per connectar dos mòduls entre si, i tota la lògica referent a aquell mòdul es troba dins del mateix mòdul, fent possible que el canvi d'un mòdul per un altre amb una implementació diferent no alteri el comportament dels altres. Per exemple, canviar una unitat de debug no farà que el control de la CPU sigui diferent.

Per tant, a l'hora de crear el mòdul PCU s'ha seguit òbviament amb aquest disseny. De les unitats anteriors (DC, IC, DMMU, IMMU, DU, CPU) s'han obtingut les senyals necessàries per poder obtenir els esdeveniments i totes les senyals han estat connectades al mòdul PCU. Això significa que no s'ha calculat la senyal per obtenir l'esdeveniment a dins de cada mòdul i a PCU només entren els esdeveniments, sinó que s'han connectat a PCU les senyals que amb certa lògica poden determinar si està succeint aquell esdeveniment o no, i li correspon al propi mòdul PCU obtenir els esdeveniments i realitzar el comptatge. La Il·lustració 5-1 mostra la ubicació del nou mòdul, amb els mòduls que s'interconnecta, al mòdul pare "or1200_top".



Il·lustració 5-1. Processador OR1200: mòdul "or1200_top" amb PCU

5.3 Registres del mòdul PCU

Els registres que conformen el mòdul PCU són tots registres de propòsit especial i només són accessibles a través de les instruccions d'accés a registres especials: `l.mtspr` i `l.mfspr`.

Recordant la Taula 3-1 sobre els grups de registres de propòsit especial, del grup 0, el grup de registres de sistema (d'estat i de control), hi ha dos registres importants que ja estan implementats però que s'han d'adaptar els valors acord amb la nova implementació.

5.3.1 Registres de configuració, grup SPR 0

El registre 1 d'aquest grup, el UPR, és un registre de configuració (només de lectura) que indica les unitats presents a la implementació actual (les caches, les MMUs, la unitat de debug, el controlador programable d'interrupcions...). El bit "PCUP" d'aquest registre correspon a si la unitat de comptadors està present o no. Per tant, s'ha d'activar a 1.

Igualment, el registre 8 d'aquest grup correspon al PCCFGR, el registre de configuració dels comptadors hardware (només de lectura). Aquest registre únicament té un conjunt de bits etiquetats com "NPC" que corresponen al nombre de comptadors disponibles al

processador. Tota la resta són marcats com a reservats.

Tal com estava especificat al manual de l'arquitectura, eren 3 bits que el valor 0b000 corresponia a 1 únic comptador, i el valor 0b111 corresponia als 8 comptadors presents (recordar que l'arquitectura defineix obligatòriament 8 comptadors). Entre mig hi havia els altres possibles valors, de 1 a 8.

No obstant, s'ha cregut convenient modificar l'especificació d'aquest registre per fer-la més coherent amb la seva pròpia descripció. Per això, s'ha agafat un dels bits reservats i s'ha estès els "NPC" a 4 bits. Amb aquest canvi, es considera que el valor 0x0 correspon a cap comptador implementat, el valor 0x1 a 1 comptador implementat, i així successivament fins a 0x8, que són 8 comptadors implementats. Cal adonar-se que a l'utilitzar 4 bits, sobren 7 valors que no estan associats. Això no és un problema perquè el registre és només de lectura i, per això, quan s'escriu el seu contingut és que s'està tocant directament el codi font, i per tant s'és conscient de la implementació desitjada.

Aquest canvi s'ha considerat molt important, perquè si no s'implementa el mòdul de comptadors, o està implementat i es desactiven tots els comptadors, el registre PCCFGR és consultable igualment (en lectura). Per tant, per mantenir la coherència, els "NPC" haurien de considerar que el valor "0" correspon a cap comptador, i no indicar que un únic comptador està present, sense que ni tan sols el mòdul estigui implementat.

Bit	31-4	3-0
Identifier	Reserved	NPC
Reset	-	-
R/W	R	R

NPC	Number of Performance Counters 0 No performance counters 1 One performance counter ... 8 Eight performance counters

Il·lustració 5-2. Descripció dels bits del registre PCCFGR (PCCFGR Field Descriptions)

La Il·lustració 5-2 mostra com ha quedat modificat el registre per poder ser incorporat de nou al manual de l'arquitectura (en anglès).

5.3.2 Registres PCCR0-PCCR7

Els registres PCCRn (*Performance Counters Count Registers*) són els registres pertanyents al grup 7, del registre 0 al 7. Aquests registres emmagatzemen el comptatge i, per tant, són els 8 comptadors dels esdeveniments que es programin. Els 32 bits del registre representen el valor sense que hi hagi control de desbordament. El programador ha de ser conscient dels límits d'aquest comptador i saber si comptarà un rang més gran que 2^{32}-1.

Il·lustració 5-3. Registre PCCR

El manual descriu que aquests registres es poden llegir des de mode usuari si el bit "SUMRA" del registre de supervisió SR (del grup 0) està activat. En la implementació actual del processador aquest bit en aquest registre no està implementat (i en lloc seu hi ha el "TED" (*trap exception disable*)). Així doncs, aquesta característica no s'ha tingut en compte perquè igualment no es podria definir al registre SR.

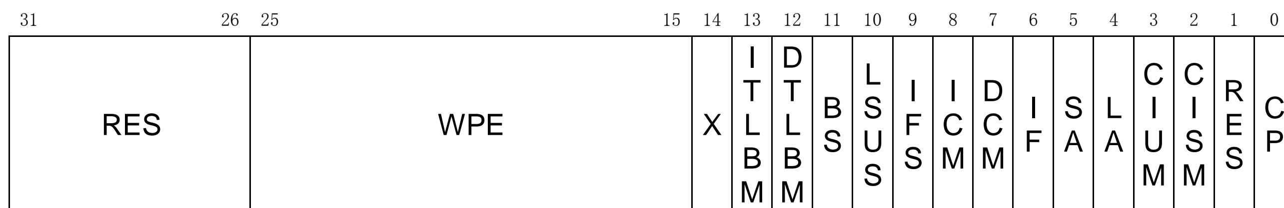
5.3.3 Registres PCMR0-PCMR7

Els registres PCMRn (*Performance Counters Mode Registers*) són els registres pertanyents al grup 7, del registre 8 al 15. Defineixen quins esdeveniments comptarà la seva parella dels registres PCCR. És a dir, la configuració establerta en el PCMR0 actuarà únicament sobre el comptador PCCR0, el PCMR3 al PCCR3, etc.

L'especificació d'aquest mòdul permet activar cada bit corresponent a un esdeveniment i això significa que un únic comptador podria comptar tots els esdeveniments de cop (cosa totalment absurda). En canvi, existeixen situacions on tenir activat més d'un esdeveniment en un sol comptador pot ser interessant, com per exemple, activar el SA i LA alhora permetrà saber tots els accessos a memòria.

És per això que s'ha implementat d'acord amb aquesta especificació deixant clar que l'usuari serà l'encarregat de saber quins bits activa i a responsabilitat seva saber què vol comptar (com comptar missos de cache + *fetch* d'instruccions + aturades per salts...).

La Il·lustració 5-4 mostra els bits del registre PCMR. Hi ha dues regions marcades com a reservades “RES”. Del bit 4 al bit 25 es poden observar els diferents esdeveniments on el valor 1 indica comptar esdeveniment i 0 no comptar-lo. El bit 14 corresponia a l’esdeveniment DDS, que no s’ha implementat. El camp “CIUM” correspon a si aquell esdeveniment està permès que sigui comptat en mode usuari i semblantment, el camp “CISM” indica si l’esdeveniment està permès que sigui comptat en mode sistema (supervisor). El bit 0 “CP” és un bit únicament de lectura que indica si aquell determinat comptador és present a l’actual implementació o no.

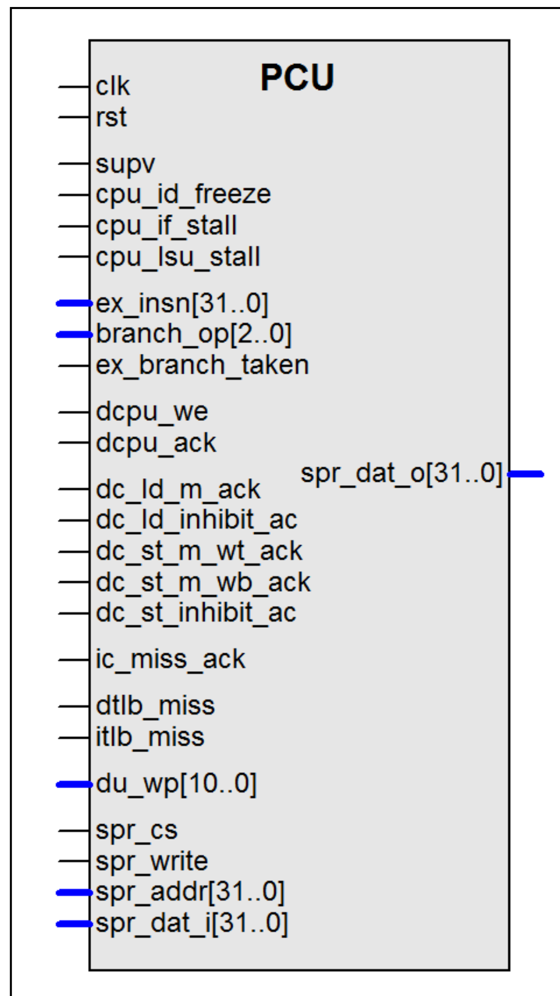


Il·lustració 5-4. Registre PCMR

Els 11 bits que conformen els “WPE” corresponen a cada un dels watchpoints de la unitat de debug. En aquest sentit, el valor 0b000_0000_0000 indica que tots els esdeveniments dels watchpoints són ignorats; el valor 0b000_0000_0001 indica que el watchpoint 0 és comptat i així successivament fins el valor 0b111_1111_1111 on tots els 11 watchpoints són comptats.

5.4 La lògica del mòdul PCU

Un cop s’ha analitzat quins esdeveniments es compten, quins mòduls generen les senyals necessàries per obtenir els esdeveniments, la ubicació d’aquest mòdul i conèixer els registres que conformen el mòdul de comptadors hardware, és hora ja d’implementar la seva lògica. La Il·lustració 5-5 mostra la interfície d’entrada i sortida del mòdul PCU.



Il·lustració 5-5. Interfície PCU

5.4.1 Implementació dels esdeveniments

Esdeveniments LA i SA

La senyal ‘`dcpu_ack`’ indica que s’ha realitzat amb èxit un accés a memòria de dades ACK (*acknowledgement* des de memòria, un únic cicle). En aquest mateix cicle, la senyal ‘`dcpu_we`’ que es correspon al permís d’escriptura de la memòria de dades, segueix tenint el valor vàlid de l’accés que s’acaba de realitzar. Per tant, cada cop que s’activi la senyal ‘`dcpu_ack`’, si la senyal ‘`dcpu_we`’ està activada sabrem que hi ha hagut un accés de store, i si no està activada, és que l’accés era de load. Això activarà directament les senyals d’esdeveniments ‘`sa_evnt`’ i ‘`la_evnt`’ que indicaran si s’ha produït un esdeveniment d’aquest tipus o no, respectivament.

Esdeveniment IF

Es pot considerar que s'ha fet *fetch* d'una instrucció sempre que el permís d'escriptura del registre per passar de l'etapa IF a ID estigui actiu. Aquest permís és la senyal 'cpu_id_freeze' negada ja que mentre està activa, les etapes del pipeline es mantenen amb la mateixa instrucció. No obstant, s'ha de comprovar que estigui passant una instrucció útil i no es tracti de la propagació d'una NOP. Per això, amb aquesta senyal negada, s'ha de fer una operació AND amb el resultat de comparar la instrucció amb una NOP. El resultat de tot això resulta amb la senyal 'if_evnt'.

Esdeveniments IFS i LSUS

La CPU del mateix processador ofereix unes senyals internes, 'cpu_if_stall' i 'cpu_lsu_stall', que serveixen directament com a les senyals d'esdeveniments 'ifs_evnt' i 'lsus_evnt' respectivament.

La senyal 'cpu_if_stall' s'activa cada cop que obté un nou PC i es desactiva quan arriba l'ACK de la memòria d'instruccions, conseqüentment, indica els cicles que s'està fent el *fetch* d'una instrucció.

Per la seva banda, la senyal 'cpu_lsu_stall' prové directament de la unitat LSU i s'activa quan una instrucció d'accés a memòria es troba a l'etapa EX i es desactiva quan arriba l'ACK de la memòria de dades. Aquesta senyal indica que la LSU està ocupada durant aquests cicles.

Esdeveniments WPE

El bus 'du_wp' s'ha extret directament de la unitat de debug on aquest és intern i activa el bit corresponent al watchpoint que s'ha produït. Per tant, aquest bus és directament també el de l'esdeveniment 'wp_evnt'. Això significa, per exemple, que si el 'wp_evnt[5]' està actiu, s'ha activat el watchpoint nº5.

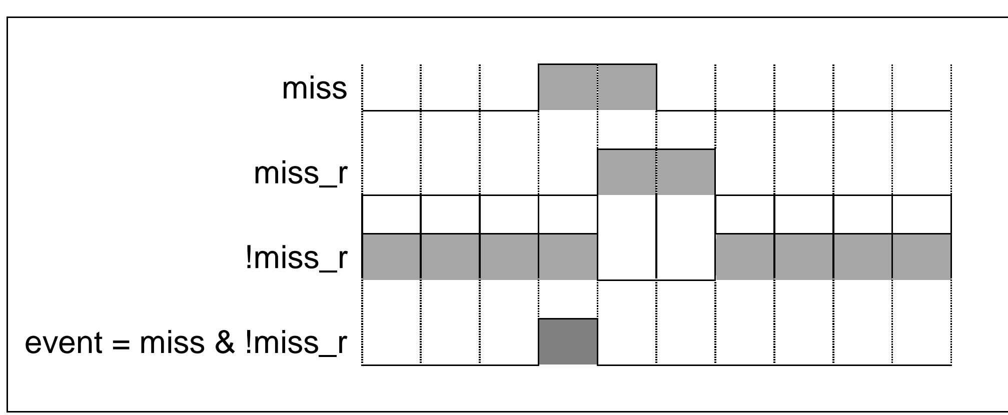
Esdeveniments ICM i DCM

La cache d'instruccions té una senyal interna 'ic_miss_ack' que indica que s'ha produït un miss. Aquesta senyal és directament la senyal d'esdeveniment 'icm_evnt'.

Pel que fa a la cache de dades, el resultat és el mateix, però la senyal està desglossada en 5. Aquestes senyals són segons si s'ha produït un miss degut a un load, un store en configuració de write-through o un store en configuració de write-back (senyals 'dc_ld_m_ack', 'dc_st_m_wt_ack' i 'dc_st_m_wb_ack' respectivament). A més, també hi ha dues senyals més que ocasionen miss (per fer les cinc): 'dc_ls_inhibit_ack' i 'dc_st_inhibit_ack'. Aquestes senyals s'activen quan la MMU de dades està habilitada i s'ha configurat una determinada pàgina perquè no vagi a cache i s'obligui a accedir a memòria principal. En aquests casos, si la cache està activada, això es tradueix, en definitiva, com a falles de cache. Per tant, una operació OR d'aquestes 5 senyals obtenen directament la senyal d'esdeveniment 'dcm_evnt'.

Esdeveniments ITLBM i DTLBM

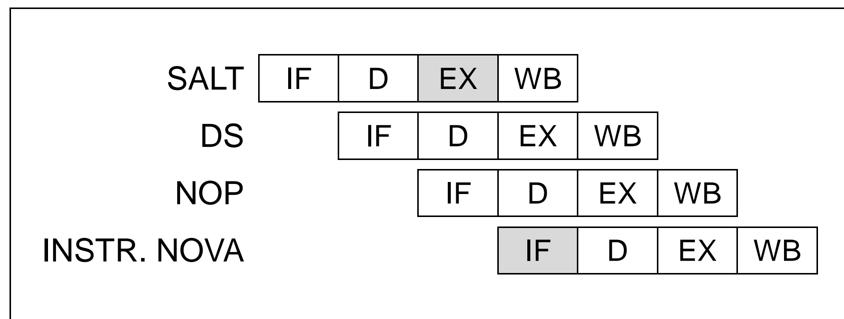
Les senyals 'itlb_miss' i 'dtlb_miss' provinents de les respectives MMUs indiquen que hi ha un miss però no l'ACK, sinó que es mantenen actives durant més d'un cicle tractant-se únicament d'un mateix miss. Per això, les senyals d'esdeveniment 'itlbm_evnt' i 'dtlb_evnt' són el resultat d'aconseguir un únic cicle per miss. Una forma senzilla ha estat registrar les dues senyals, i fer una operació AND amb la senyal actual i la registrada negada, tal com mostra el cronograma de la Il·lustració 5-6.



Il·lustració 5-6. Cronograma d'esdeveniment de fallades de TLB

Esdeveniment BS

En el moment en què un salt arriba a l'etapa EX se sabrà si el salt es realitza o no (serà *taken* o no). A partir de llavors, es carrega el nou PC i es va a memòria a buscar la instrucció. Al cicle següent doncs, quan el salt estigui a l'etapa WB, la nova instrucció estarà en l'etapa IF. Això significa que hi ha dos cicles entre el salt i la nova instrucció (entre l'execució del salt i de la nova instrucció) com es pot veure a la Il·lustració 5-7.



Il·lustració 5-7.Pipeline durant un salt

Durant aquests dos cicles, el processador propagaria NOPs, però l'OR1200 té implementat un *delay slot* (DS) on en temps de compilació d'un programa hi posa una instrucció vàlida que s'hauria d'haver executat abans que el salt. Per tant, la instrucció que s'executa en DS és vàlida i no s'insereix/propaga una NOP. No obstant, només hi ha un DS i per tant, al següent cicle sí que es propaga la NOP.

Això significa que com a màxim dels 2 cicles, les aturades degut a salt seran aquells cicles d'execució en els quals s'hagi inserit una NOP degut al salt. Per fer això, es registra la senyal '`ex_branch_taken`' mentre el processador no estigui aturat i durant dos cicles. Llavors si aquestes senyals resultants (la del primer cicle o la del segon) estan activades, es compara la instrucció en execució '`ex_insn`' amb l'identificador de la NOP i això determinarà la senyal d'esdeveniment '`bs_evnt`'.

Únicament les instruccions de tipus "*jump*" i les de tipus "*branch*" són considerades com a salt. Ara bé, existeix un paràmetre al fitxer de configuració (or1200_defines.v) que permet canviar aquesta opció i fer que es comptin totes les aturades degut a instruccions que trenquen la seqüenciació implícita, com poden ser retorns d'excepcions, la instrucció `l.trap`, *breakpoints* software, etc.

5.4.2 Implementació dels registres comptadors

Tant els registres PCCR com els PCMR s'accedeixen a través de la interfície de registres especials. Cada registre correspon a una adreça i pot ser escrit o llegit amb les instruccions "`l.mtspr`" i "`l.mfspr`". Tal i com es defineixen aquestes instruccions, la suma del registre font més el literal formen l'adreça, mapejada a cada registre.

Per llegir existeix un gran descodificador, que envia pel bus 'spr_dat_o' el contingut del registre segons encaixi amb l'adreça 'spr_addr'. Per escriure, s'espera a tenir permís d'escriptura amb la senyal 'spr_write' i igualment segons l'adreça es realitza l'escriptura al registre al següent flanc ascendent de la senyal de rellotge.

Al fitxer de configuració (or1200_defines.v) existeix un altre paràmetre, activat per defecte, per determinar si es vol llegir els bits no utilitzats, reservats i no implementats com a 0 o no. Per exemple, el bits reservats del PCMR, al llegir el registre, es retornaran com si continguessin el valor 0 i no un valor indeterminat.

Pel que fa a la detecció d'esdeveniments, ja s'ha vist que el registre PCMR pot tenir habilitats més d'un esdeveniment a la vegada per a un únic comptador. El que significa això és que un comptador no s'incrementa en una unitat, sinó que es podria donar la situació, poc probable però no impossible, que dos o més esdeveniments estiguessin activats al mateix cicle, i per tant, al cas extrem, incrementar el comptador amb una quantitat igual al nombre màxim d'esdeveniments.

Per complir amb aquesta situació, s'ha implementat la solució següent per cada parella de comptadors PCMR-PCCR:

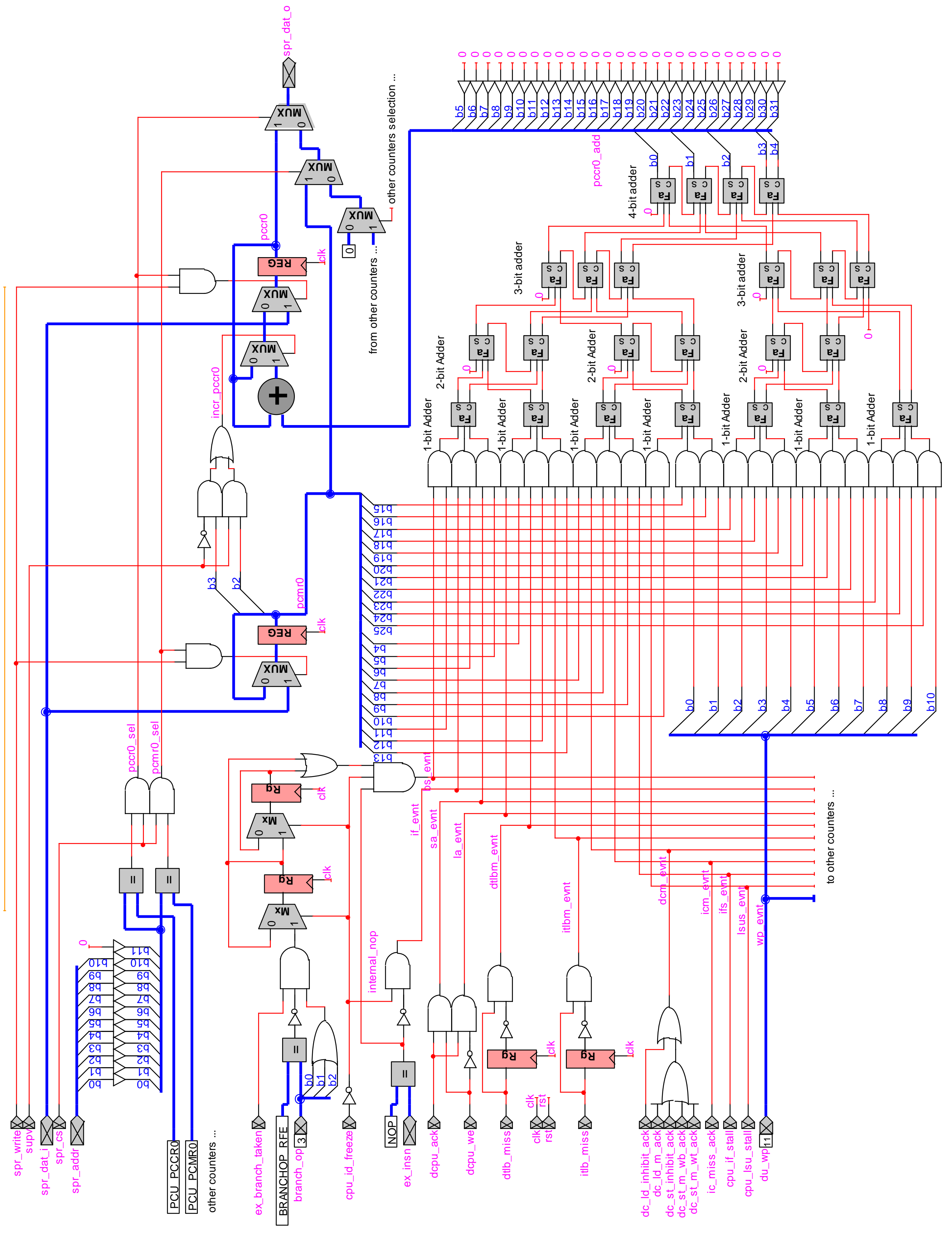
- S'ha comparat cada bit del registre PCMR corresponent a un esdeveniment, amb cada senyal d'esdeveniment. Per exemple, PCMR[IF] amb la senyal 'if_evnt'.
- Cada resultat del pas anterior s'ha sumat un amb l'altre per obtenir un bus que s'ha considerat l'incrementador del comptador (anomenat 'pccr_add').
- Finalment, el comptador s'ha incrementat amb 'pccr_add' si la senyal 'incr_pccr' està activada.

La senyal 'incr_pccr' es pot considerar, per cada comptador, com el permís d'escriptura per incrementar-lo. En concret, aquesta senyal només està activada si el processador està en mode usuari i el bit "CIUM" del registre PCMR està activat, o si el processador està en mode sistema i el bit "CISM" del registre PCMR està activat. Per poder saber en quin mode està actualment el processador, al mòdul entra la senyal 'supv' que es correspon amb el bit SM del registre de propòsit especial SR: Si el valor del bit és 1 el processador està en mode sistema, i si per contra és 0, el processador està en mode usuari.

Per acabar, un comptador pot començar a comptar esdeveniments al cicle següent que s'habiliti el comptatge a PCMR. En aquest moment, si en aquest mateix cicle que ja està activat es detecta l'esdeveniment a comptar; al cicle següent, amb el flanc ascendent de la senyal de rellotge, es realitza l'escriptura, amb el primer increment.

La Il·lustració 5-8 mostra la implementació final del mòdul de comptadors hardware a circuit lògic. A la part esquerra (en l'orientació que apareix el text dins la imatge) es pot apreciar la detecció dels esdeveniments tal com s'han descrit, a la part central la lògica que controla quins esdeveniments s'han de comptar, i a la part dreta l'obtenció de l'increment a realitzar.

Tot això es mostra per a un únic comptador, la parella PCCR0-PCMR0, però per obtenir els dels altres s'ha de replicar aquest mateix circuit.

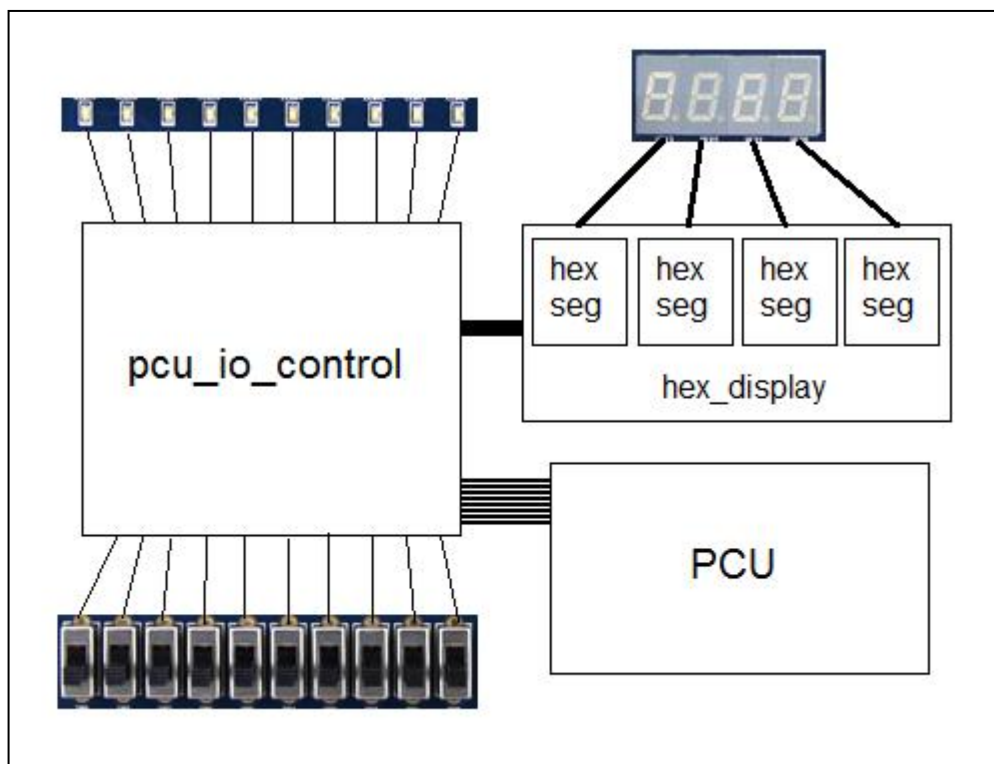


Il·lustració 5-8. Circuit lògic del mòdul PCU

5.5 Implementació d'un visualitzador de comptadors extern al processador

Amb el mòdul de comptadors implementat, ja aconseguim tenir de nou el processador funcional. L'objectiu final és que pugui executar-se sobre la FPGA, però el resultat d'una execució queda camuflat dins del processador, i és necessari disposar de la unitat de debug i el depurador GDB per poder obtenir els resultats.

De totes maneres, disposem d'una placa DE1 amb molts dispositius terminals. Per tant, s'ha decidit implementar un petit mòdul, com a *core* per a la DE1, que controli la visualització dels comptadors i puguem veure'n el resultat pels 4 visors de 7 segments disponibles a la placa durant la mateixa execució d'aquests.



Il·lustració 5-9. Connexió pcu_io_control amb la resta d'elements

Per això, s'han connectat els switchs, els leds vermells i els 4 visors amb un controlador que obté el contingut dels registres PCCR dels comptadors hardware a temps real, tal com mostra la Il·lustració 5-9.

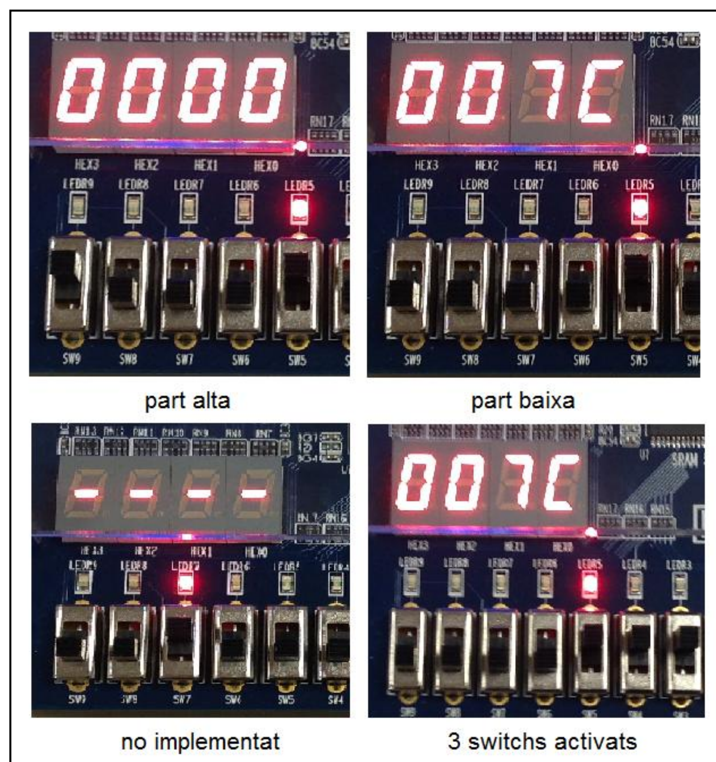
S'ha mapejat cada switch de 0 a 7 amb el contingut del registre PCCR del 0 al 7 respectivament. Un cop seleccionat, el contingut del registre és enviat al mòdul

“hex_display” que separa el nombre en 4 dígits hexadecimals, que són enviats als visors. Com que els visors poden mostrar un nombre fins a 16 bits, i els comptadors són de 32 bits, el switch 9 s’ha fet servir com a la selecció de la part alta del comptador [31..16] (si està activat, posició cap amunt) o la part baixa [15..0] (si està desactivat, posició cap avall).

Com que els switchs són mecànics i se’n poden activar més d’un a la vegada, com a criteri el programa mostra el contingut del switch de més pes: si estan activats els switchs 5 i 7, es mostrarà el contingut del comptador nº7. Per facilitar saber quin és el comptador actualment visible, a més a més, també s’han utilitzat els leds vermells, on el led encès és el nombre de comptador que s’està mostrant pels visors.

Finalment, com que es pot donar el cas que un determinat comptador no estigui implementat, en comptes de mostrar el valor “0000” (que correspon a que el comptador no s’ha incrementat), apareixen 4 guions “- - - -”.

La Il·lustració 5-10 mostra els casos descrits.

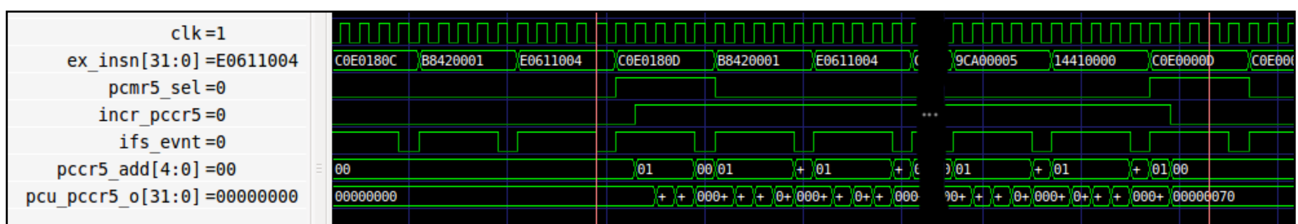


Il·lustració 5-10. Demostració dels casos descrits (pcu_io_control)

6 RESULTATS

Amb el mòdul de comptadors implementat i abans de programar-lo físicament a la placa, seguint la metodologia del projecte, ha calgut testejar i comprovar que funciona correctament en simulació.

Per això, s'ha creat un joc de proves específic per cada esdeveniment i s'ha provat diferents configuracions dels registres PCMR per comprovar que el resultat és el desitjat. A l'Annex D es pot trobar el joc de proves íntegre escrit en llenguatge assemblador amb el joc d'instruccions ORBIS32 d'OpenRISC i la descripció de cada joc.



Il·lustració 6-1. Captura de pantalla d'una simulació

S'han analitzat les senyals a través de la simulació com mostra la Il·lustració 6-1 corresponent a una captura d'una de les simulacions, i s'han detectat errors que han estat corregits.

Un dels errors importants, era el càlcul de l'esdeveniment IF, els *fetch* d'instruccions. Resulta que primerament només s'havia contemplat que l'esdeveniment era possible sempre que el permís d'escriptura del registre entre l'etapa IF i ID estigués actiu, però el resultat era que al comptador apareixia un nombre més gran que les instruccions que formaven el joc de proves. Gràcies a la simulació, i analitzant cada cop que s'incrementava el comptador quines eren les instruccions que passaven per aquell registre, es va poder veure que les instruccions de més eren les NOP que s'inserien al pipeline i que no s'havien tingut en compte. Llavors va ser quan es va modificar l'esdeveniment perquè no considerés les NOP com a *fetch* d'instruccions.

D'altra banda també hi havia errors menors, com que el PCCR2 no comptava bé, perquè mirava alguns bits de configuració del PCMR1 i no els de la seva parella, corresponent al PCMR2.

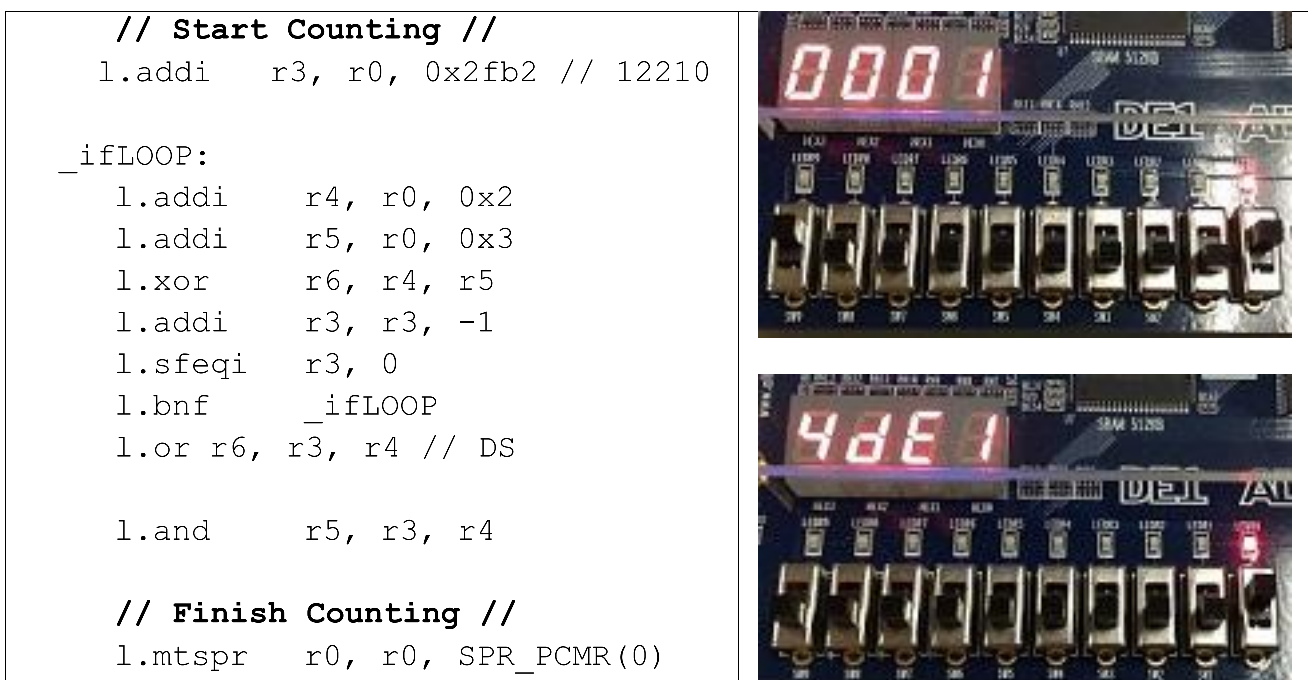
Finalment, veient que els resultats eren coherents s’ha procedit a programar el processador a la placa, a passar el joc de proves i obtenir una execució en viu.

Cal tenir present que els jocs de prova no asseguren al 100% el correcte funcionament de la implementació, però es realitzen suficients casos i proves que si són satisfactòries es considera que la implementació és correcta.

Per demostrar el correcte funcionament, a continuació es detalla únicament la porció de codi que es compta durant els esdeveniments, mostrant 5 dels 12 tests del joc de proves de l’Annex D, s’analitza teòricament el valor esperat i es compara amb una fotografia del resultat d’executar allò mateix a la placa física.

Els únics esdeveniments que no han estat provats amb jocs de prova han estat els WPE, els watchpoints de la unitat de debug perquè en la configuració utilitzada pel processador estaven deshabilitats. No obstat, s’ha assignat (en simulació) que el bus d’esdeveniments dels watchpoints prengués valors per comprovar que incrementaven correctament.

Esdeveniment IF



```
     // Start Counting //
    l.addi   r3, r0, 0x2fb2 // 12210

  _ifLOOP:
     l.addi    r4, r0, 0x2
     l.addi    r5, r0, 0x3
     l.xor     r6, r4, r5
     l.addi    r3, r3, -1
     l.sfeqi   r3, 0
     l.bnf     _ifLOOP
     l.or r6, r3, r4 // DS

     l.and     r5, r3, r4

     // Finish Counting //
     l.mtspr   r0, r0, SPR_PCMR(0)
```

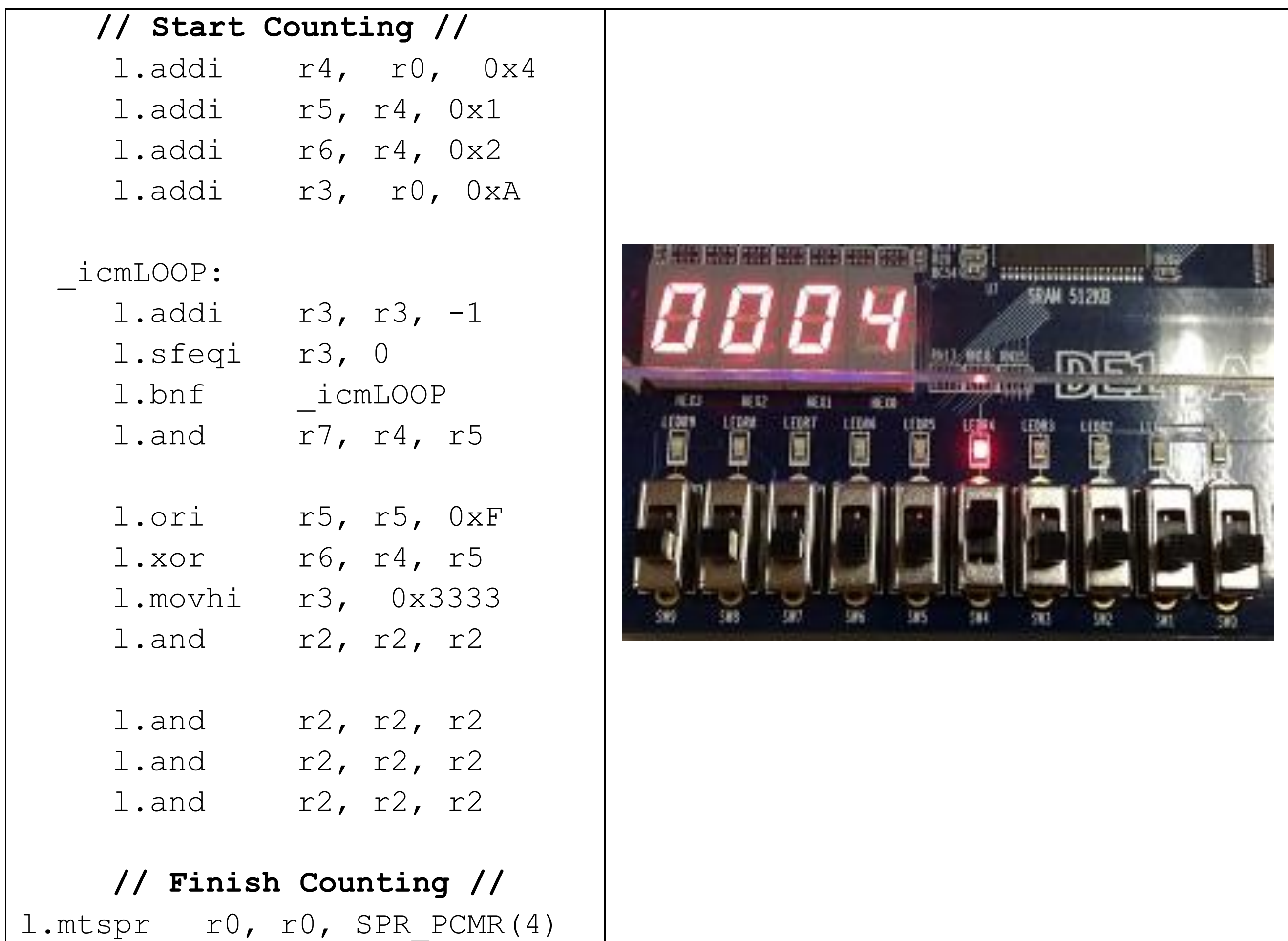

Taula 6-1. Joc de prova esdeveniment IF

El resultat es correspon al *fetch* de la primera instrucció posterior a l’activació del comptador + el bucle + la instrucció *and* i també la instrucció que atura el comptador,

doncs quan aturi el comptador ja haurà fet el *fetch* d'aquesta. És a dir: 1(addi) + 7 instr. * 12210 iteracions + 1(and) + 1(mtspr) = 85470 = 0x14de1. Notar que el 0x1 de la part alta es mostra separat a la imatge.

Esdeveniment ICM

La cache d'instruccions és de 4Kb, amb 256 línies de 16 bytes cada una. Això significa que a cada línia hi caben 4 instruccions.



```
    // Start Counting //
     l.addi    r4,  r0,  0x4
     l.addi    r5, r4, 0x1
     l.addi    r6, r4, 0x2
     l.addi    r3,  r0, 0xA

  _icmLOOP:
     l.addi    r3, r3, -1
     l.sfeqi   r3, 0
     l.bnf     _icmLOOP
     l.and     r7, r4, r5

     l.ori     r5, r5, 0xF
     l.xor     r6, r4, r5
     l.movhi   r3,  0x3333
     l.and     r2, r2, r2

     l.and     r2, r2, r2
     l.and     r2, r2, r2
     l.and     r2, r2, r2

     // Finish Counting //
l.mtspr   r0, r0, SPR_PCMR(4)
```

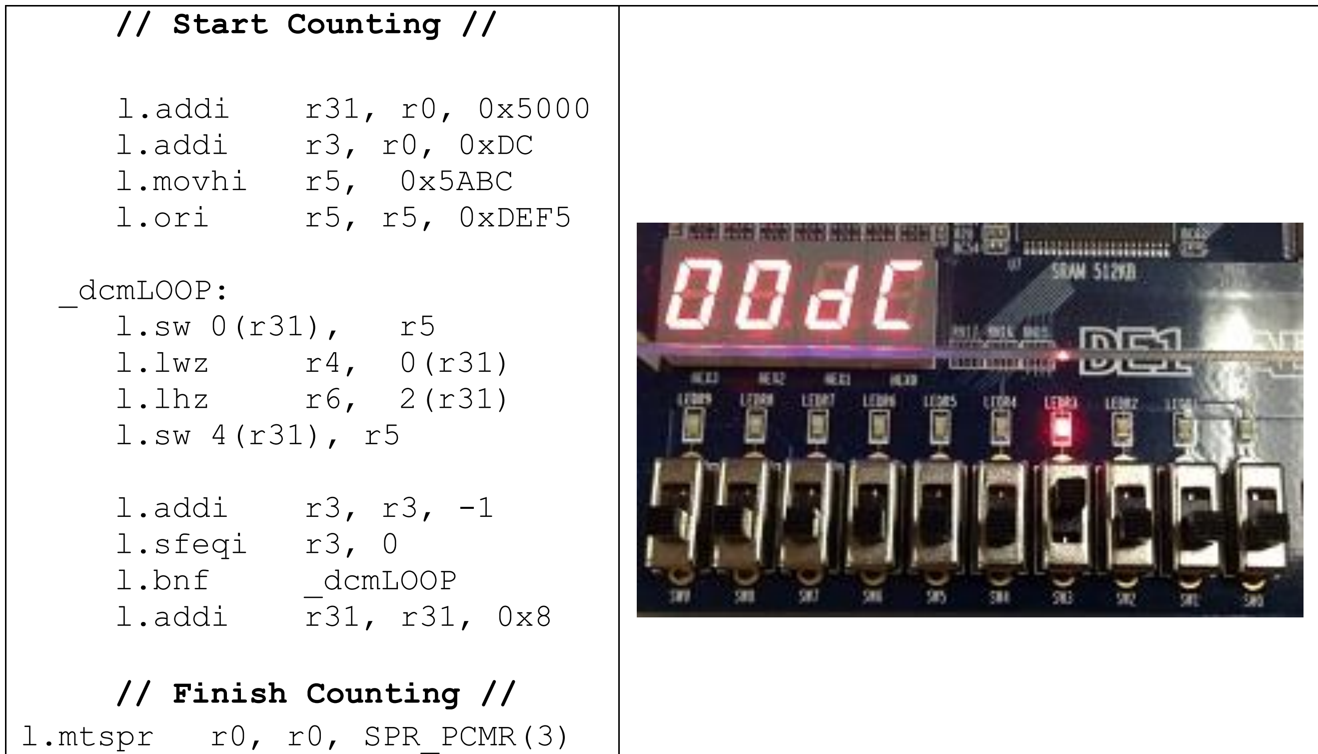

Taula 6-2. Joc de prova esdeveniment ICM

Com es pot veure hi ha 4 grups de 4 instruccions (la instrucció que atura el comptador també es compta ja que ja s'ha fet el *fetch* d'aquesta instrucció, i per tant, s'ha portat de memòria). La consideració especial és fixar-se que hi ha un bucle de 10 iteracions, però que únicament fallarà un cop, el primer cop que entri al bucle, perquè a partir de llavors les instruccions ja estaran a la cache. Per tant, si cada 4 instruccions una falla, i tenim 4 grups de 4 instruccions, el resultat és de 4 missos.

Esdeveniment DCM

La cache de dades és de 4Kb, amb 256 línies de 16 bytes cada una, amb política Write-through.

```
     // Start Counting //

     l.addi    r31, r0, 0x5000
     l.addi    r3, r0, 0xDC
     l.movhi   r5,  0x5ABC
     l.ori     r5, r5, 0xDEF5

  _dcmLOOP:
     l.sw 0(r31),   r5
     l.lwz     r4,  0(r31)
     l.lhz     r6,  2(r31)
     l.sw 4(r31), r5

     l.addi    r3, r3, -1
     l.sfeqi   r3, 0
     l.bnf     _dcmLOOP
     l.addi    r31, r31, 0x8

     // Finish Counting //
l.mtspr   r0, r0, SPR_PCMR(3)
```



Taula 6-3. Joc de prova esdeveniment ICM

El registre r31 serveix d'adreça base. El registre r3 conté les 220 iteracions (0xDC). A r5 hi ha la dada a emmagatzemar.

Com que no hi ha la línia a la cache, el primer store falla i per la política utilitzada no es porta la línia a la cache. Després es fa un load que també falla però porta la línia, i a la següent iteració s'accedeix als altres 8 bytes de la mateixa línia, seguint cíclicament cada 2 iteracions el mateix patró: *Miss – Miss – Hit – Hit – Hit – Hit – Hit – Hit*.

Per tant, el resultat serà els 2 missos cada 2 iteracions: 2M / 2 * 220 iteracions = 220 = 0xDC.

Esdeveniment LA+SA

En aquest joc de proves es comptaran dos esdeveniment alhora, i a més es comprovarà que només es compta segons els bits que indiquen si es permet comptar en mode usuari i en mode sistema.

La configuració del registre no es mostra en aquesta descripció, però es correspon a comptar únicament en mode usuari. L'estat actual, quan comença el joc de proves és en mode sistema.

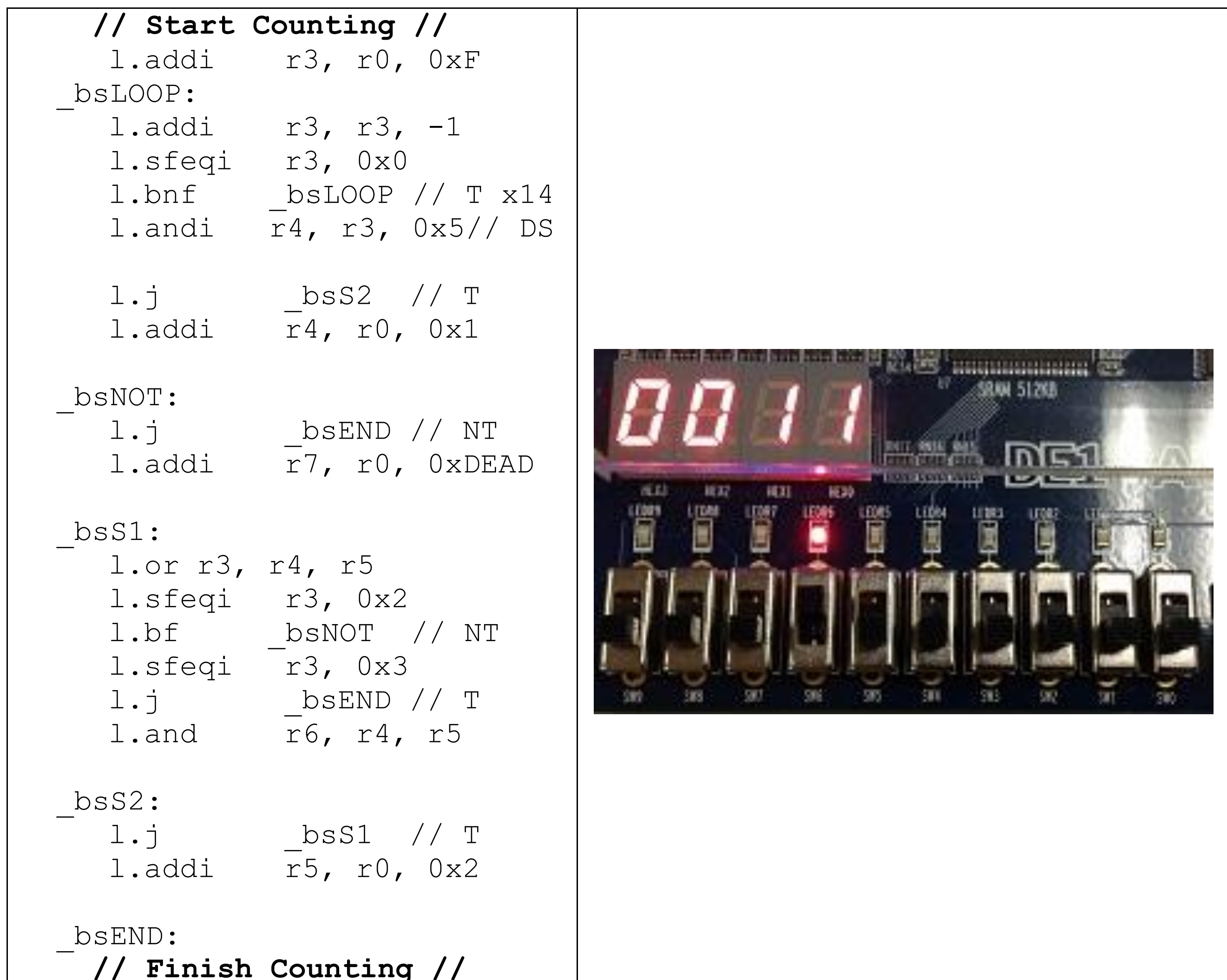
```
// Start Counting //
 // CPU = SM
 l.sw 0x0(r4), r5
 l.sw 0x0(r4), r5
 l.lwz     r8, 0x0(r4)

 // Put CPU in User Mode
 l.mfspr   r2, r0, SPR_SR
 l.and     r2, r2, r1
 l.mtspr   r0, r2, SPR_SR
 // CPU = UM
 l.sw 0x4(r4), r6
 l.sw 0x8(r4), r7
 l.sw 0xC(r4), r8
 l.lwz     r7, 0x4(r4)
 l.lwz     r6, 0x8(r4)
 l.lwz     r5, 0xC(r4)

 // Trigger sys call
 l.sys     0
 // CPU = SM
 // Finish Counting //
```

Taula 6-4. Joc de prova esdeveniments LA i SA amb comprovació del bit CIUM

Comptant els loads i store, veiem que n'hi ha 3 que es fan en mode sistema i 6 en mode usuari. Correctament podem veure que el valor resultant correspon als accessos quan s'estava en mode usuari.

Cal tenir en compte que no es pot accedir als registres especials si no s'està en mode sistema i que, per tant, no es pot desactivar el comptador sense tornar a mode sistema. Igualment, el registre SR deixa d'estar visible. Cadrà fer una crida a sistema per tornar-hi.

Esdeveniment BS




```
  // Start Counting //
   l.addi    r3, r0, 0xF
 _bsLOOP:
   l.addi    r3, r3, -1
   l.sfeqi   r3, 0x0
   l.bnf    _bsLOOP // T x14
   l.andi   r4, r3, 0x5// DS

   l.j       _bsS2  // T
   l.addi    r4, r0, 0x1

 _bsNOT:
   l.j       _bsEND // NT
   l.addi    r7, r0, 0xDEAD

 _bsS1:
   l.or r3, r4, r5
   l.sfeqi   r3, 0x2
   l.bf     _bsNOT  // NT
   l.sfeqi   r3, 0x3
   l.j       _bsEND // T
   l.and     r6, r4, r5

 _bsS2:
   l.j       _bsS1  // T
   l.addi    r5, r0, 0x2

 _bsEND:
  // Finish Counting //
```

Taula 6-5. Joc de prova esdeveniment BS

Quan un salt està en etapa d'execució i és *taken* (T), o sigui, que salta, la nova instrucció no estarà disponible fins 2 cicles més tard. En el primer, hi ha una instrucció vàlida al *delay slot* (DS). Al segon, ja no n'hi ha més i una NOP s'insereix. Per tant, a cada salt *taken* una NOP s'insereix al pipeline i es perd 1 cicle.

En el joc anterior, es realitza un bucle de 15 iteracions, que saltarà a totes les iteracions excepte la última, per continuar amb el codi. A partir d'aquí es fan un seguits de salts sense saltar al "_bsNOT". Així doncs, al finalitzar el programa, s'han fet 14 salts degut al bucle, i 3 més fins a acabar ("_bsS1", "_bsS2" i "_bsEND"). Per tant, el resultat és 14 + 3 = 17, que en hexadecimal són 0x11.

6.1 Impacte del bus i de la memòria principal en els resultats

Amb certs jocs de prova que s'han provat s'han detectat certes ocurrències que no són errors però sí que poden fer que el resultat d'un comptador varïi segons quina sigui l'execució. En concret, els esdeveniments que es veuen més afectats són els que compten cicles d'aturada, com l'IFS o el LSUS.

Les aturades degut a que s'està cercant a memòria la següent instrucció, corresponen (sense disposar de cache d'instruccions activada) a:

- 8 cicles d'aturada on es llegeixen les instruccions de memòria i es porten al controlador.
- A partir de llavors, cada 4 cicles el controlador serveix una instrucció pel bus *Wishbone*. Això significa que s'atura 3 cicles i un altre és d'execució.
- Quan s'acaben les 8 instruccions, torna a començar de nou un altre període.

El que passa però, és que la memòria que s'utilitza a la placa és la SDRAM, i per tant, hi ha cicles ocults (entre el controlador i la memòria) com per exemple, per al refresc de la memòria, o carregar el *row buffer* amb la fila pertinent perquè la línia actual no és l'activada, etc.

Similarment, les aturades de LSU poden variar degut a la memòria i a la gestió que es faci entre el controlador de la SDRAM i l'arbitre del bus *Whisbone*.

Els valors resultats que mostren aquests comptadors poden, llavors, no encaixar amb el codi que s'estigui executant, però si s'analitzen les senyal ja sigui a través d'una simulació o amb un analitzador lògic, es podrà comprovar que són correctes.

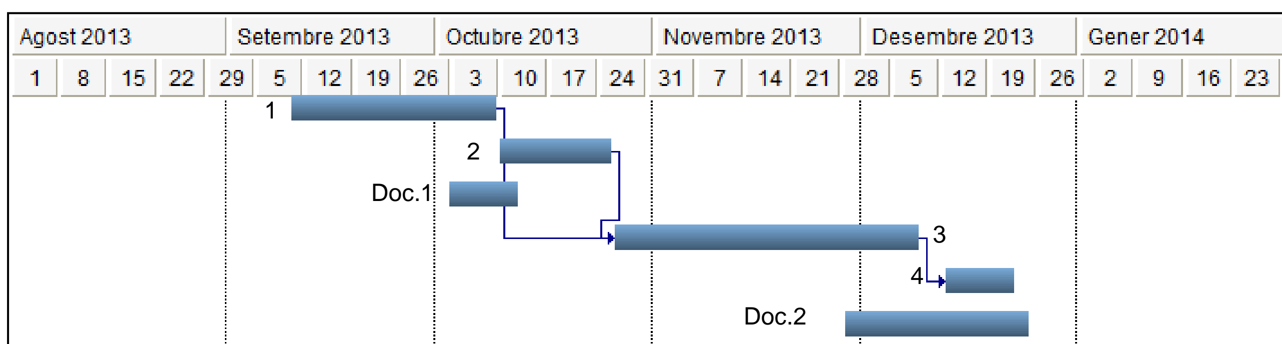
Finalment, cal tenir present que, per exemple, si es compten fallades de cache d'instruccions però no es posa un nombre d'instruccions múltiple de 4 (segons la configuració de la cache d'instruccions a la placa DE1), el resultat del comptador pot variar en un factor de 1, segons el nombre d'instruccions que ja s'hagin carregat prèviament a la cache.

7 DESENVOLUPAMENT DEL PROJECTE

7.1 Planificació

7.1.1 Planificació inicial

Al mes de juliol es va definir una planificació del que esdevindria el projecte. Les etapes inicials eren les que es mostren al diagrama de Gantt següent.



Il·lustració 7-1. Diagrama de Gantt inicial

<u>Tasca 1: Entendre l'arquitectura i el processador OpenRISC</u>

Aquesta tasca consisteix en familiaritzar-se amb l'entorn. Es llegirà el document amb l'especificació i l'arquitectura d'OpenRISC i s'analitzarà el codi font del processador per detectar les etapes de segmentació, amb quines senyals i busos es transmet cada dada a través de les etapes, com funciona el control del processador i com es gestionen els riscs al processador. S'acabarà generant un esquema gràcies a tota la informació obtinguda per una visió ràpida de com està dissenyat el processador (mostrant les etapes de segmentació).

Període: Del 9 de setembre de 2013 al 7 d'octubre de 2013 (18 dies hàbils).

Recursos:

- Manual d'arquitectura OpenRISC
- Codi font del processador OpenRISC 1200. S'utilitzarà la versió *System On Chip* (OrpSoCv2) ja que la intenció és utilitzar una FPGA amb components externs (pulsadors, leds, etc.)

Tasca 2: Configuració de l'estació de treball

S'aconseguirà una estació de treball adequada per tal de poder modificar el codi, generar els fitxers compilats per a la FPGA, simular el comportament del processador a nivell software únicament, i disposar de totes les eines útils de la ToolChain d'OpenRISC (*compiler, debugger*) i simulador. Es considerarà acabada aquesta tasca en el moment que es pugui simular el processador base (sense cap modificació) i es pugui executar sobre la FPGA.

Període: Del 8 d'octubre de 2013 al 23 d'octubre de 2013 (12 dies hàbils).

Recursos:

- Placa de desenvolupament DE1 d'Altera que incorpora una FPGA
- Programa Quartus II d'Altera: Per a la compilació del codi i realitzar el *placement, routing...* així com programar la FPGA
- Simulador ModelSim integració amb Altera per a visualitzar el comportament del processador
- ToolChain d'OpenRISC

Tasca 3: Implementació i comprovació

Implementació dels comptadors hardware en el processador OpenRISC 1200. Aquesta tasca és la d'implementar els comptadors seguint l'especificació definida. A mida que es vagi implementant s'anirà comprovant el correcte funcionament (descrit a metodologia).

Aquesta tasca no podrà començar fins que s'hagin completat les dues anteriors. Això permetrà una ràpida comprovació perquè per una banda, tot el software ja estarà configurat adequadament, i per l'altra, s'intentarà minimitzar el nombre d'errors que puguin sorgir ja que es tindrà un coneixement de tot el processador en conjunt abans no es comenci a modificar el codi (d'aquí la primera tasca).

Període: Del 24 d'octubre de 2013 al 5 de desembre de 2013 (30 dies hàbils).

Tasca 4: Generació de resultats

Dins del marge de temps d'aquesta tasca es generaran els resultats després d'haver implementat els comptadors hardware. S'obtindran resultats sobre diversos jocs de prova o programes reals per demostrar el funcionament dels comptadors hardware.

Període: Del 9 de desembre al 18 de desembre de 2013 (8 dies hàbils).

Tasca 5: Documentació

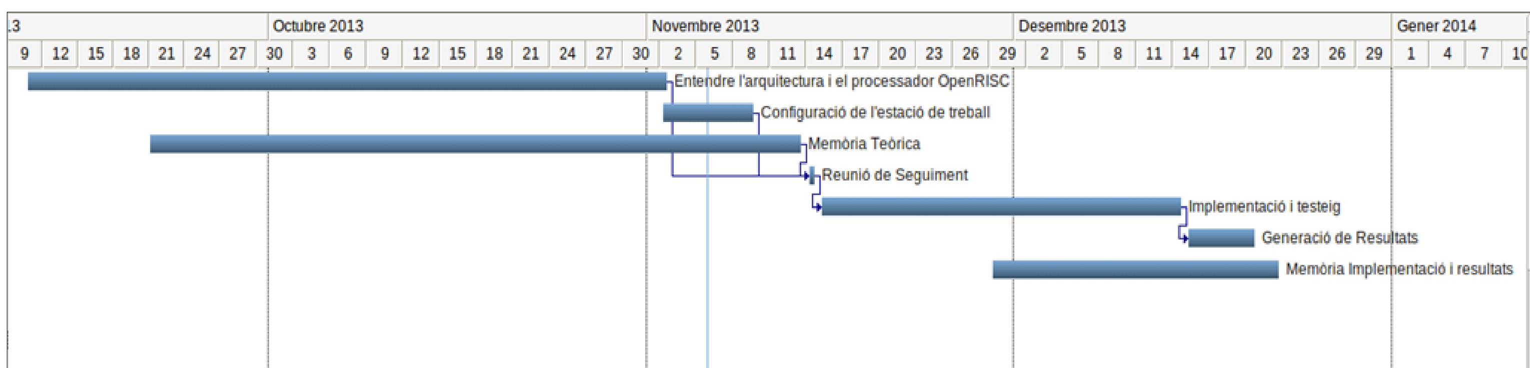
Aquest apartat conté les dues tasques anomenades documentació 1 i 2. Es tracta de la redacció de la memòria del projecte així com esquemes i dades resultats de tot el projecte. S'ha dividit al llarg del projecte perquè un cop es coneix l'arquitectura del processador, és un bon moment per redactar-ho, així es va creant el que serà el document final d'entrada i ajudar a mantenir els conceptes per a tasques posteriors com la d'implementació.

Quan s'estigui cap a la part final d'implementació ja es tindrà un coneixement de tot el projecte, que és quan se seguirà redactant fins a enllestir el projecte. No és un problema que se sobreposin aquestes últimes tasques, ja que cada cop que s'obtingui un resultat o una millora, ja es pot anar anotant, deixant dos dies hàbils per acabar de polir els últims detalls de la memòria.

Període: De l'1 d'octubre de 2013 al 10 d'octubre de 2013 com a inici i del 25 de novembre de 2013 al 20 de desembre de 2013 (27 dies hàbils).

7.1.2 Canvi de planificació

Al començar setembre, en una reunió amb el director, es va decidir modificar la planificació inicial. La nova reestructuració és mostrada al següent diagrama de Gantt.



Il·lustració 7-2. Diagrama de Gantt (Setembre)

Les tasques segueixen essent les mateix, però s’ha modificat la durada d’aquestes. En concret, es va decidir que tota la part teòrica del treball, és a dir, tota la documentació del processador i la redacció de la memòria dels conceptes teòrics a tractar es tindria acabada per la reunió de seguiment, el 12 de novembre de 2013. A partir de llavors, es

dedicaria únicament al disseny i a la implementació dels comptadors hardware al processador, descriure la seva descripció a la memòria i la generació dels resultats.

Els canvis més significatius són el d'estendre la tasca de “Documentació 1”, que s'ha reanomenat a “Memòria Teòrica”, per tractar tota la part teòrica del projecte, i que s'encavalca amb la tasca d'entendre l'arquitectura d'OpenRISC i la seva documentació, on s'ha dibuixat la circuiteria a partir del codi font del processador, i s'han redactat els temes teòrics de la memòria. I la tasca descrita com “Documentació 2” a la planificació inicial, que s'ha reanomenat com “Memòria Implementació i resultats”.

7.1.3 Planificació real

Amb el canvi de planificació del setembre, les tasques inicials fins a la reunió de seguiment s’han acomplert amb èxit i durant el temps establert: S’ha estudiat el processador, s’ha desenvolupat la documentació necessària i s’ha escrit la part teòrica de la memòria, corresponent fins el capítol 4, inclòs.

A partir de la reunió de seguiment però, les dues tasques finals no s’han acomplert amb la previsió de durada considerades.

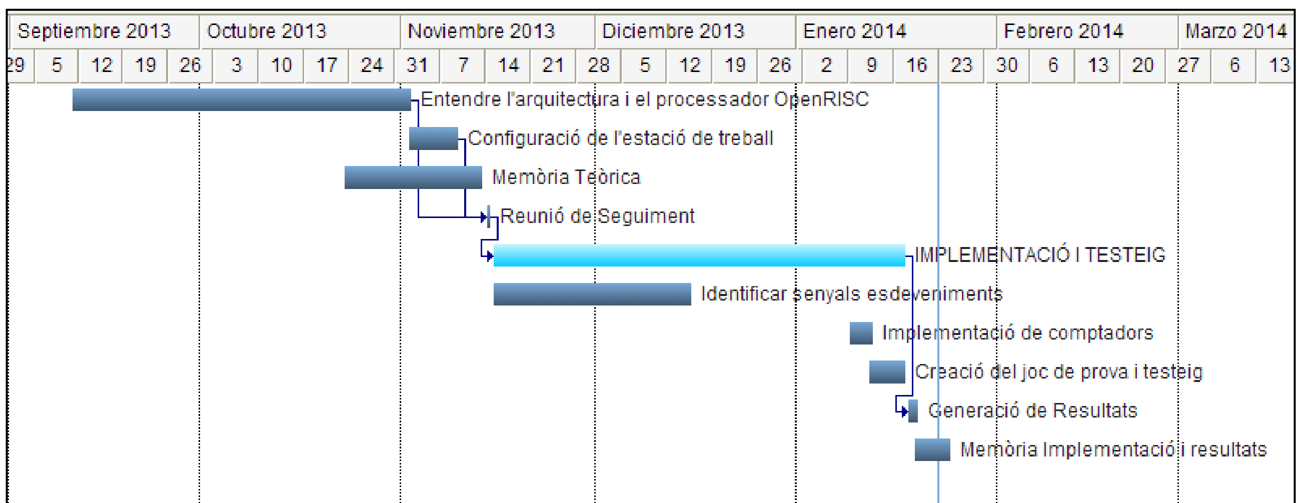
Es considerava que des del 13 de novembre es començaria amb l’etapa d’implementació i que s’acabaria el 12 de desembre. Doncs bé, aquesta tasca ha estat la que ha fet retardar l’etapa de resultats.

Implementar no només significava escriure el codi perquè els comptadors funcionessin, sinó que també s’ha correspost a analitzar i saber quines senyals del processador es podien utilitzar per comptar els esdeveniments descrits.

El procés d’identificació d’aquestes senyals, i la seva posterior comprovació perquè realment servissin per al nostre ús, a base de moltes simulacions del comportament del processador, ha estat més llarg de l’esperat. Després s’ha implementat els comptadors i finalment s’ha creat un joc de proves extens per comprovar el seu funcionament.

En concret, tota aquesta fase d'implementació i testeig ha ocupat 29 dels 21 dies lectius que havia de durar. En ubicació mensual s'ha traduït des del 13 de novembre de 2013 al 15 de gener de 2014. Per motius personals, la última setmana de desembre (del 16 al 20) no es va poder elaborar projecte i durant les festes de Nadal no s'ha avançat, fent aquests 29 dies lectius.

A partir d'aquí, s'han generat els resultats, la demostració que els comptadors funcionen sobre la placa, i s'ha escrit la segona part de la memòria, per finalitzar-la. El diagrama de Gantt final següent mostra l'evolució real del projecte.



Il·lustració 7-3. Diagrama de Gantt de la planificació real

7.2 Cost del projecte

El cost total del projecte és representat per les diferents llicències dels programes utilitzats, de la placa de desenvolupament i la feina del desenvolupador.

Com que es tracta d'un projecte orientat en un entorn acadèmic i desenvolupat per a aquest, el cost de les llicències són gratuïtes ja que s'utilitzen les versions educatives o acadèmiques per als programes de pagament (Quartus II, ModelSIM i Microsoft Office).

El programa Logic Works 5, amb el qual s'ha dibuixat tota la circuiteria d'OR1200, no disposa de llicència educativa i el software ve adjunt amb el llibre que porta el mateix nom. El cost d'aquest llibre és de 73€.

Tota la resta de programari és de codi lliure i per tant, gratuït.

En un any, es poden treballar 1800h. Aquest càlcul surt de saber que l'any té 365 dies, dels quals 104 dies no són laborables (52 caps de setmana), 22 són dies de vacances (que acaben de completar al mes, doncs els caps de setmana ja s'han descomptat), 12 dies festius oficials i 2 dies per a festes locals. Per tant, queden 225 dies laborables. Determinant que es treballa 8 hores cada dia, 8h*225dies = 1800h/any.

El projecte de TFG consta de 15 crèdits ECTS i 1 crèdit ECTS equival a 30 hores. Per tant, el projecte consta de 15*30h = 450h.

Comptant una amortització de 2 anys per al software, el programa Logic Works costa pel seu ús: 73€ * (450h/82 dies projecte * 36 dies (tasca 1)) / (1800h/any * 2 anys amort.) = **4€**

El cost de la placa DE1 presenta un descompte per a un entorn acadèmic, costant 96,77€. Així doncs, comptant una amortització del hardware de 3 anys: 96,77€ * (450h/82 dies * 46 dies (excepte tasca 1)) / (1800h/any * 3anys amort.) = **4,52€**

Finalment, cal tenir present el cost del desenvolupador. Partint que els directors del projecte em van dir que un desenvolupador de HDL cobra uns 30.000€ a l'any, podem assignar un preu per hora de 30.000€ / 1800h = 16.6€ -> 17€/h. Així doncs, 450h / 82 dies * 73 dies (entendre, implementar, testejar) * 17€/h = **6810,36€**

Destacar que al preu de desenvolupador no s'han comptat l'IRPF ni d'altres impostos que a l'empresa li suposarà al voltant del 30% més.

A part, tampoc s'han comptabilitzat despeses generals com electricitat, connexió telefònica, etc., perquè el projecte s'ha desenvolupat en un entorn acadèmic. Si no fos aquest el cas, llavors caldria saber el cost de les despeses generals i establir altre cop la relació per hores.

La Taula 7-1 mostra el resum de costos de cada recurs i el total.

Recurs	PREU
Llicència Altera Quartus II Web Edition	0€
Llicència ModelSIM Altera Starter-Edition	0€
ToolChain d'OpenRISC	0€
Simulador Icarus Verilog	0€
Visualitzador GTK Wave	0€
Microsoft Office	0€
Logic Works 5	4€
Placa DE1 Development and Education board	4,52€
Desenvolupador	6810,36€
TOTAL:	**6818,88€**

Taula 7-1. Cost total del projecte

7.3 Sostenibilitat i compromís social

El fet d'haver escollit utilitzar la placa DE1 com a entorn físic on provar el processador, amb una FPGA, ve donat pel fet que el resultat obtingut un cop s'ha programat és semblant al que s'obtindria després de crear el xip real en qüestió, ja que es tracta d'un dispositiu hardware, però amb l'avantatge que no cal gastar temps, recursos ni matèries per construir-lo físicament.

A més a més, el fet de tractar-se d'un element programable obre un ventall de possibilitats perquè aquestes plaques podran ser utilitzades per a moltes altres finalitats, assignatures, etc., perquè cadascú hi podrà programar el comportament que desitgi en qualsevol moment, significant que aquesta placa no estigui limitada en exclusiu a aquest projecte.

La durabilitat està garantida perquè les plaques vénen protegides amb un plàstic dur transparent, evitant que s'hi puguin donar cops i per tant, ajudant a la seva preservació durant anys.

Finalment i com a factor humà, l'elecció d'una placa com la DE1 ha sigut pensant també en les persones que la utilitzaran. En principi va destinada als estudiants i es valora positivament que amb tots els components perifèrics de què disposa serveixin perquè hi hagi interacció amb l'estudiant i per tant, l'aprenentatge sigui més fructífer, aspecte clau de qualsevol docent.

7.4 Lleis i regulacions

El projecte està destinat a un ús docent i no estan involucrats usuaris externs. Així doncs, les lleis que afecten el projecte són les lligades a l'ús, modificació i distribució que es pot fer del codi i de l'arquitectura amb la qual s'ha treballat, l'OpenRISC.

Les llicències que afecten al projecte OpenRISC són les següents:

- L'especificació de l'arquitectura està publicada sota una llicència GPL (*GNU General Public License*), que permet a un usuari la llibertat d'utilitzar, modificar, compartir i millorar l'objecte sota aquesta llicència.
- El codi Verilog del processador està publicat sota una llicència LGPL (*GNU Lesser General Public License*), que consisteix bàsicament amb el mateix criteri que la GPL però permet que pugui ser utilitzat a més, per programes que no estiguin sota una llicència lliure.

Això significa doncs, que un cop acabat el projecte, el codi d'OpenRISC seguirà estant sota aquesta llicència LGPL i per tant, podrà ser compartida i utilitzada per altres usuaris a disposició del codi lliure.

7.5 Competències tècniques

A continuació es detallen les competències tècniques associades al projecte segons el grau de profunditat i com s'han tractat al llarg del projecte.

CEC1.2: *Dissenyar/configurar un circuit integrat utilitzant les eines de software adients. [En profunditat]*

Aquesta competència s'ha desenvolupat en profunditat ja que es tracta de l'objectiu final del projecte: dissenyar i implementar un mòdul de comptadors hardware en el processador OR1200. S'ha determinat com havien de ser els esdeveniments que ja venien establers, s'ha implementat el mòdul i s'ha programat sobre una FPGA per comprovar-ne la seva execució sobre un entorn físic real.

Per acomplir amb èxit aquesta part, ha calgut disposar de les eines software pertinents, com el simulador per analitzar el comportament de les diferents senyals, el programari d'Altera per programar la FPGA de la placa DE1 i els compiladors d'OpenRISC.

CEC2.1: *Analitzar, avaluar, seleccionar i configurar plataformes hardware per al desenvolupament i l'execució d'aplicacions i serveis informàtics. [Bastant]*

Aquesta competència s'ha utilitzat durant les primeres fases del projecte, amb l'anàlisi de processadors existents i la posterior selecció d'OpenRISC.

Vinculada amb aquesta competència es pot considerar l'extracció dels diferents circuits a partir del codi font del processador OR1200 per analitzar el processador i obtenir el pipeline d'aquest.

8 CONCLUSIONS

Tot i la desviació que ha presentat el projecte, d'un allargament de 12 dies corresponents a una durada de 82 dies respecte dels 70 establers a la planificació inicial, es pot concloure que el projecte s'ha realitzat amb èxit i s'han acomplert els objectius.

En el primer d'ells, avaluar diferents processadors, a apartat 2.2 s'ha mostrat una taula amb les característiques més importants de cada un i s'ha justificat el criteri de selecció.

En el segon, s'ha generat tota una documentació sobre OpenRISC desglossada entre el capítol 3 i l'Annex C.

Seguint amb els objectius, s'ha implementat el mòdul de comptadors hardware per a OpenRISC i s'ha creat tot un gran joc de proves específic per provar el mòdul. Com es pot apreciar al capítol de resultats, l'execució d'aquest mòdul és correcta.

Com a possible treball futur, es pot analitzar el consum que provoca cada esdeveniment i contrastar el consum total amb el calculat amb els comptadors.

Una altra tasca a realitzar pot ser inserir el mòdul com a codi font permanent dins el projecte OpenRISC. Per això però, caldria obtenir la nova versió ORPSoCv3 (que com s'ha indicat al projecte durant la realització d'aquest ha sortit la nova versió i s'ha decidit no canviar-la) afegir-hi els comptadors i entregar la feina al projecte OpenRISC per a una revisió i possible incorporació final.

9 BIBLIOGRAFIA

[1] S. Palnitkar, Verilog HDL: A guide to Digital Design and Synthesis, Cap. 1, Segona ed., Prentice Hall, 2008.

[2] ARM, «Cortex-M1 Processor,» [En línia] <http://www.arm.com/products/processors/cortex-m/cortex-m1.php> [Últim accés: Octubre 2013].

[3] Z. Gomez, «Zet Processor,» [En línia] <http://zet.aluzina.org/> [Últim accés: Octubre 2013].

[4] OpenCores, «OpenRISC Project,» [En línia] <http://opencores.org/or1k/OR1K:Community_portal> [Últim accés: Novembre 2013].

[5] Gaisler, «LEON3 Processor,» [En línia] <http://gaisler.com/index.php/products/processors/leon3> [Últim accés: Octubre 2013].

[6] Simply RISC, «S1 Core,» [En línia] <http://www.simplyrisc.com/> [Últim accés: Octubre 2013].

[7] L. Dimitris, «OpenCores - MIPS R2000,» [En línia] <http://opencores.org/project,mipsr2000> [Últim accés: Setembre 2013].

[8] Quickwayne, «OpenCores - Performance counter for Microblaze,» [En línia] <http://opencores.org/project,performance_counter> [Últim accés: Octubre 2013].

[9] Intel, «Performance Monitoring,» de *Intel® 64 and IA-32 Architectures Software Developer's Manual*, vol. 3B, 2013, pp. 18.1-18.95.

[10] ARM, «The Performance Monitors Extension,» de *ARM® Architecture Reference Manual*, ARMv7-A and ARMv7-R ed., vol. Cb, 2012, pp. 2299-2332.

[11] Opencores, «Opencores,» [En línia] <http://opencores.org/> [Últim accés: Octubre 2013].

[12] ÅAC Microtec, «ÅAC Microtec, RIA Components - OBC lite 500,» [En línia] <http://www.aacmicrotec.com/> [Últim accés: Octubre 2013].

[13] ORSoC.se, «OpenRISC SoC FPGA development board (Ordb2a-ep4ce22),» [En línia] <http://opencores.org/or1k/Ordb2a-ep4ce22> [Últim accés: Octubre 2013].

[14] D. Lampret, Y. Vernier i J. Baxter, OpenRISC 1000 Architecture Manual, OpenCores.org, 2012.

[15] Opencores, «Opencores OR1200 Processor,» [En línia] <http://opencores.org/or1k/OR1200_OpenRISC_Processor> [Últim accés: Octubre 2013].

[16] G. R. Smith, FPGAs 101: Everything you need to know to get started, Cap. 3, Newnes, 2010.

[17] Altera, «Altera Corporation,» [En línia] <http://www.altera.com> [Últim accés: Novembre 2013].

[18] Capilano Computing Systems, «LogicWorks 5,» [En línia] <http://www.capilano.com/lww_5> [Últim accés: Novembre 2013].

[19] S. Williams, «Icarus Verilog,» [En línia] <http://iverilog.icarus.com/> [Últim accés: Novembre 2013].

[20] J. Wheeler i T. Bybell, «GTKWave,» [En línia] <http://gtkwave.sourceforge.net/> [Últim accés: Novembre 2013].

[21] J. Bennett i J. Baxter, «OpenRISC GNU tool chain,» [En línia] <http://opencores.org/or1k/OpenRISC_GNU_tool_chain> [Últim accés: Novembre 2013].

[22] J. Baxter i J. Bennett, «Or1ksim - OR1K architectural simulator,» [En línia] <http://opencores.org/or1k/Or1ksim> [Últim accés: Novembre 2013].

[23] F. Jullien, «or1k-tcltools en Github,» [En línia] <https://github.com/fjullien/or1k-tcltools> [Últim accés: Novembre 2013].

ANNEX A MATERIAL ADJUNT

Juntament amb la memòria s'ha entregat un directori amb material que complementa aquest treball. A continuació es detalla el seu contingut:

- 'orpsocv2_original': Aquest directori conté els fitxers font del processador en l'estat inicial, sense cap modificació.

- 'orpsocv2_pcu': Aquest directori conté els fitxers font del processador OR1200 amb la implementació correcta del mòdul PCU.

- 'codi_modificat': En aquest directori s'han replicat els fitxers modificats més importants presents a "orpsocv2_pcu" perquè siguin fàcils de trobar i analitzar. En concret hi ha el fitxer de configuració "or1200_defines.v", el fitxer amb la implementació dels comptadors "or1200_pcu.v", el *core* creat per a la visualització dels comptadors a la placa DE1 'core_pcu_de1' i el joc de proves en assemblador dels comptadors "cust-pcuall.S".

- 'circuits_or1200': Aquest directori conté els circuits creats a partir de la circuiteria del processador.

    - Dins del subdirectori 'imatges' es poden trobar totes les imatges que s'han creat, a mida real, amb una resolució més adequada que la que es pot trobar a l'Annex C. Estan classificats igualment per subdirectoris segons el mòdul al qual pertanyen. Dins d'aquest mateix subdirectori es pot trobar un directori amb màquines d'estat del mòdul d'excepcions i de les dues cache.

    - Dins del subdirectori 'LogicWorks_circuits' hi ha els circuits originals per poder obrir-se amb el programa Logic Works. Notar que només es posen els nivell superiors, ja que un cop el circuit és obert amb el programa, es pot entrar dins de cada mòdul i veure'n el seu subcircuit, fins arribar a elements bàsics.

ANNEX B Comptadors a Intel i ARM

En aquest annex es presenta un extracte del manual tant d'Intel com d'ARM per poder constatar la diferència a l'hora d'implementar comptadors.

Extracte del manual d'arquitectura d'Intel "Intel® 64 and IA-32 Architectures Software Developer's Manual ", 3B 18-3. Correspon al registre de selecció d'esdeveniments i configuracions per a un determinat comptador "x":

18.2.1.1 Architectural Performance Monitoring Version 1 Facilities

Architectural performance monitoring facilities include a set of performance monitoring counters and performance event select registers. These MSRs have the following properties:

- IA32_PMCx MSRs start at address 0C1H and occupy a contiguous block of MSR address space; the number of MSRs per logical processor is reported using CPUID.0AH:EAX[15:8].
- IA32_PERFEVTSELx MSRs start at address 186H and occupy a contiguous block of MSR address space. Each performance event select register is paired with a corresponding performance counter in the 0C1H address block.
- The bit width of an IA32_PMCx MSR is reported using the CPUID.0AH:EAX[23:16]. This the number of valid bits for read operation. On write operations, the lower-order 32 bits of the MSR may be written with any value, and the high-order bits are sign-extended from the value of bit 31.
- Bit field layout of IA32_PERFEVTSELx MSRs is defined architecturally.

See Figure 18-1 for the bit field layout of IA32_PERFEVTSELx MSRs. The bit fields are:

- **Event select field (bits 0 through 7)** — Selects the event logic unit used to detect microarchitectural conditions (see Table 18-1, for a list of architectural events and their 8-bit codes). The set of values for this field is defined architecturally; each value corresponds to an event logic unit for use with an architectural performance event. The number of architectural events is queried using CPUID.0AH:EAX. A processor may support only a subset of pre-defined values.

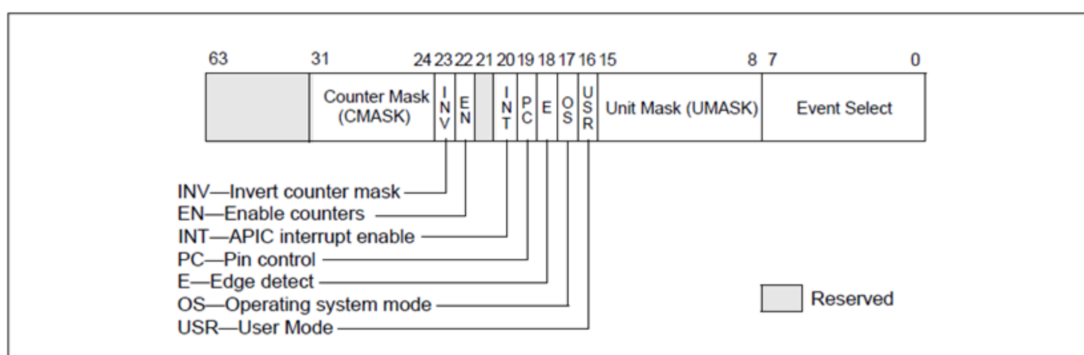


Figure 18-1. Layout of IA32_PERFEVTSELx MSRs

- **Unit mask (UMASK) field (bits 8 through 15)** — These bits qualify the condition that the selected event logic unit detects. Valid UMASK values for each event logic unit are specific to the unit. For each architectural performance event, its corresponding UMASK value defines a specific microarchitectural condition.

  A pre-defined microarchitectural condition associated with an architectural event may not be applicable to a given processor. The processor then reports only a subset of pre-defined architectural events. Pre-defined architectural events are listed in Table 18-1; support for pre-defined architectural events is enumerated using CPUID.0AH:EBX. Architectural performance events available in the initial implementation are listed in Table 19-1.
- **USR (user mode) flag (bit 16)** — Specifies that the selected microarchitectural condition is counted only when the logical processor is operating at privilege levels 1, 2 or 3. This flag can be used with the OS flag.

- **OS (operating system mode) flag (bit 17)** — Specifies that the selected microarchitectural condition is counted only when the logical processor is operating at privilege level 0. This flag can be used with the USR flag.
- **E (edge detect) flag (bit 18)** — Enables (when set) edge detection of the selected microarchitectural condition. The logical processor counts the number of deasserted to asserted transitions for any condition that can be expressed by the other fields. The mechanism does not permit back-to-back assertions to be distinguished.

  This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).
- **PC (pin control) flag (bit 19)** — When set, the logical processor toggles the PM*i* pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PM*i* pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.
- **INT (APIC interrupt enable) flag (bit 20)** — When set, the logical processor generates an exception through its local APIC on counter overflow.
- **EN (Enable Counters) Flag (bit 22)** — When set, performance counting is enabled in the corresponding performance-monitoring counter; when clear, the corresponding counter is disabled. The event logic unit for a UMASK must be disabled by setting IA32_PERFEVTSELx[bit 22] = 0, before writing to IA32_PMCx.
- **INV (invert) flag (bit 23)** — Inverts the result of the counter-mask comparison when set, so that both greater than and less than comparisons can be made.
- **Counter mask (CMASK) field (bits 24 through 31)** — When this field is not zero, a logical processor compares this mask to the events count of the detected microarchitectural condition during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented.

  This mask is intended for software to characterize microarchitectural conditions that can count multiple occurrences per cycle (for example, two or more instructions retired per clock; or bus queue occupations). If the counter-mask field is 0, then the counter is incremented each cycle by the event count associated with multiple occurrences.

Il·lustració B-1. Registre IA32_PERFEVTSELx d'Intel

Pel que fa a ARM, a la Il·lustració B-2 a continuació, hi ha l'extracte del manual d'arquitectura d'ARM "ARM® Architecture Reference Manual, ARMv7-A and ARMv7-R edition C.b", B4-1676. Correspon al registre de selecció d'esdeveniments.

En aquest cas però, no es configuren paràmetres, simplement és una selecció. Si es mira en detall l'especificació dels registres per als comptadors hardware, es veurà que ARM desglossa en registres aquestes configuracions, assignant-los a esdeveniments i comptadors.

B4.1.122 PMSELR, Performance Monitors Event Counter Selection Register, VMSA

The PMSELR characteristics are:

Purpose
- In PMUv1, PMSELR selects an event counter, PMN*x*.
- In PMUv2, PMSELR selects an event counter, PMN*x*, or the cycle counter, CCNT. The PMSELR.SEL value of 31 selects the cycle counter.

This register is a Performance Monitors register.

Usage constraints
The PMSELR is accessible in:
- all modes executing at PL1 or higher
- User mode when PMUSERENR.EN == 1.

See *Access permissions* on page C12-2328 for more information. See also *Counter access* on page C12-2312.

PMSELR is not visible in an external debug interface or a memory-mapped interface to the Performance Monitors registers.

When using CP15 to access the Performance Monitors registers, PMSELR is used in conjunction with:
- PMXEVTYPER, to determine:
  - the event that increments a selected event counter
  - in PMUv2, the modes and states in which the selected counter increments.
- PMXEVCNTR, to determine the value of a selected event counter.

Configurations
Implemented only as part of the Performance Monitors Extension.

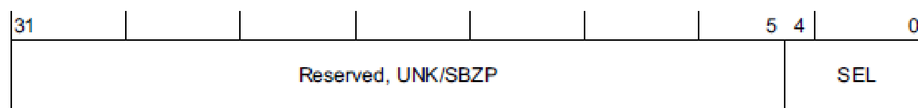
The VMSA and PMSA definitions of the register fields are identical.

In a VMSA implementation that includes the Security Extensions, this is a Common register.

Attributes
A 32-bit RW register with a reset value that is UNKNOWN on a non-debug logic reset. See also *Power domains and Performance Monitors registers reset* on page C12-2327.

Table C12-7 on page C12-2327 shows the CP15 encodings of all of the Performance Monitors registers.

The PMSELR bit assignments are:



Bits[31:5]
Reserved, UNK/SBZP.

SEL, bits[4:0]
Selects event counter, PMN*x*, where *x* is the value held in this field. That is, the SEL field identifies which event counter, PMN_{SEL}, is accessed, when a subsequent access to PMXEVTYPER or PMXEVCNTR occurs. In:

PMUv1 This field can take any value from 0 (`0b00000`) to (PMCR.N)-1. The value of `0b11111` is reserved and must not be used.

If this field is set to a value greater than or equal to the number of implemented counters the results are UNPREDICTABLE.

PMUv2 This field can take any value from 0 (`0b00000`) to (PMCR.N)-1, or 31 (`0b11111`). When PMSELR.SEL is `0b11111`:
- it selects the PMXEVTYPER for the cycle counter
- a read or write of PMXEVCNTR is UNPREDICTABLE.

If this field is set to a value greater than or equal to the number of implemented counters, but not equal to 31, the results are UNPREDICTABLE.

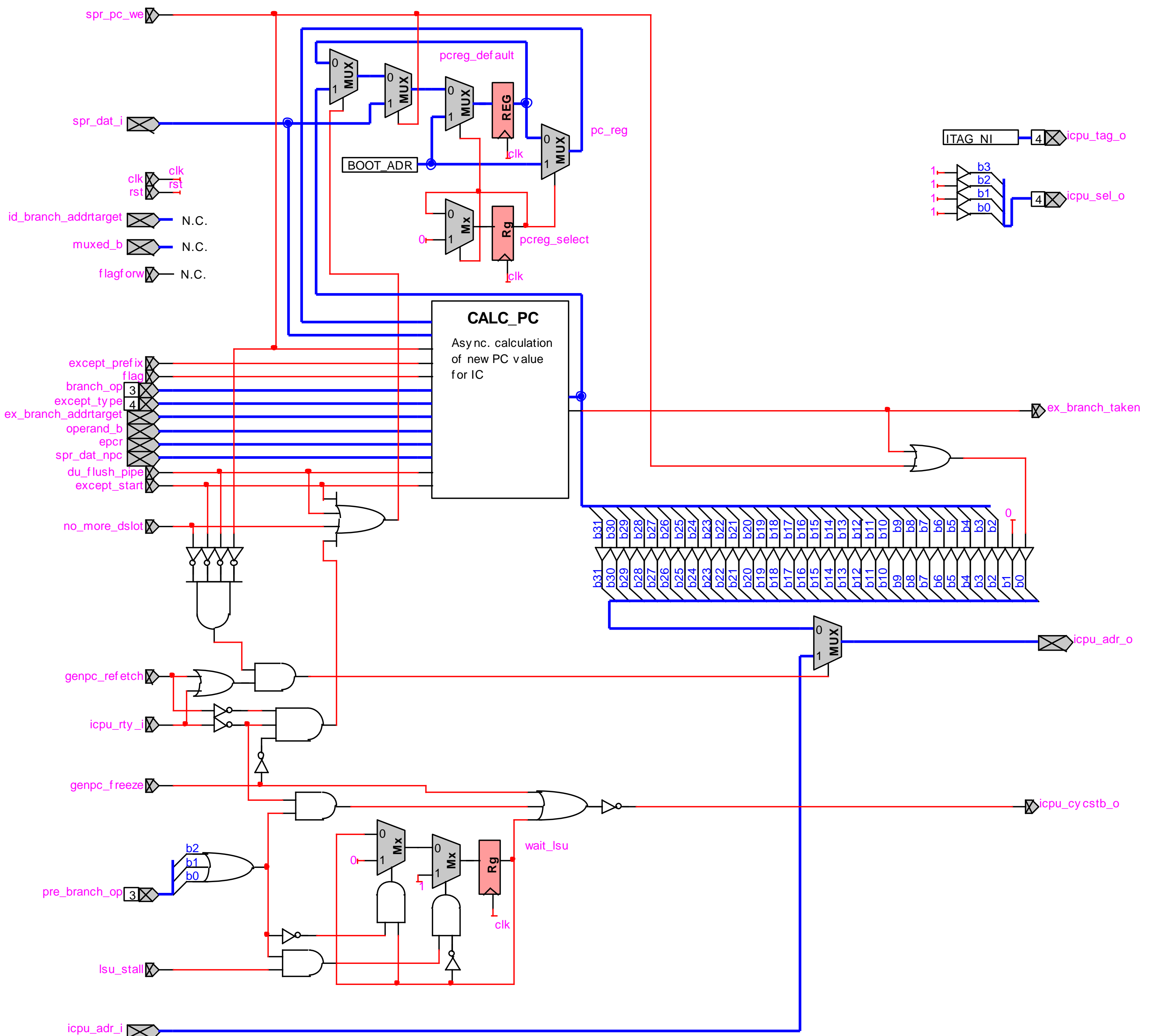
Note

PMCR.N defines the number of implemented counters.

Il·lustració B-2. Registre PMSELR d'ARM

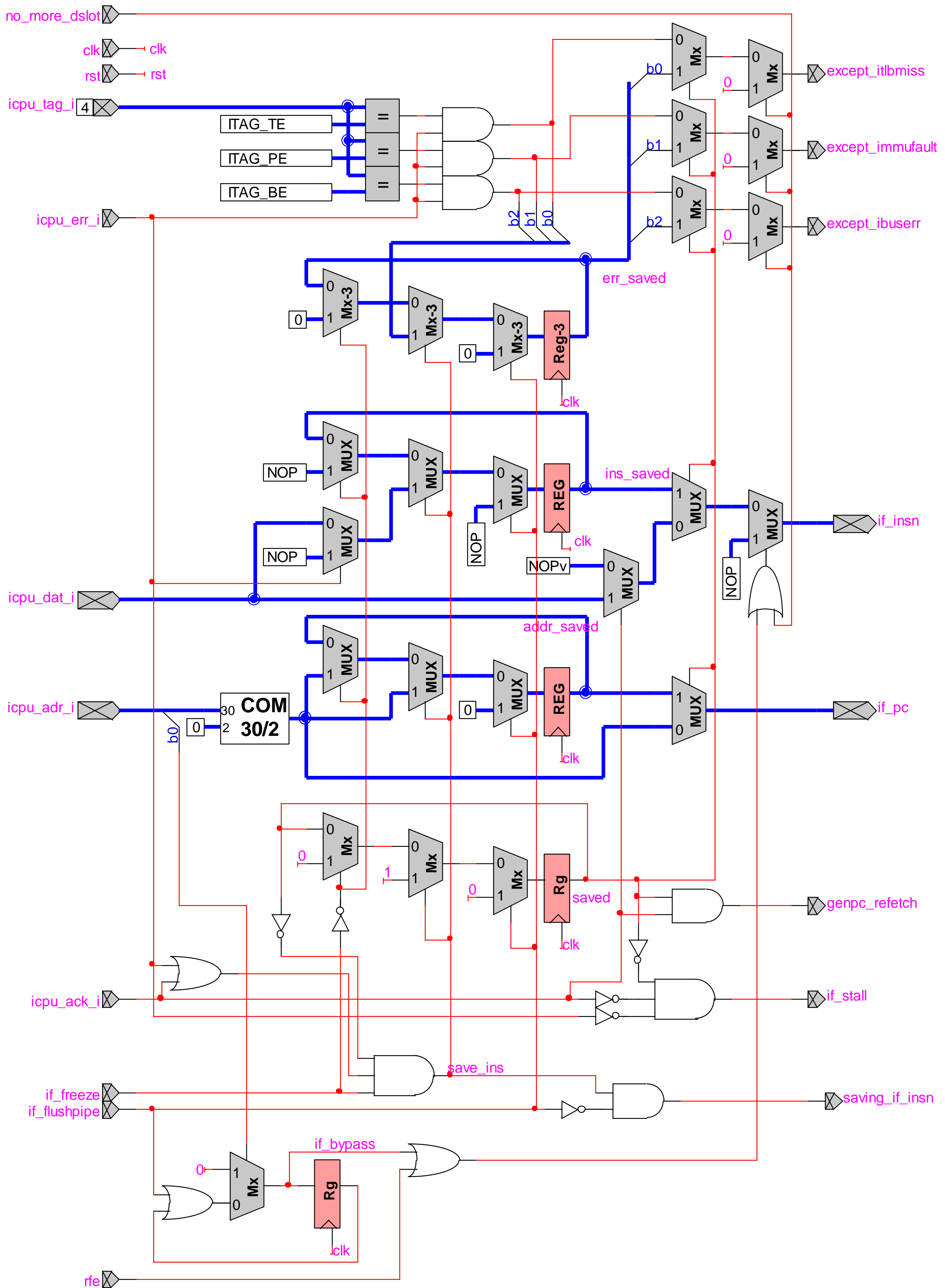
ANNEX C CIRCUITS D'OPENRISC

En aquest annex es mostren alguns dels circuits que s'han realitzat a partir del codi font del processador OR1200 d'OpenRISC per a la construcció del pipeline del capítol 3. El mòdul potser més interessant, la CPU, no s'ha posat en aquest annex per la gran mida que ocupa (es pot trobar al material adjunt).

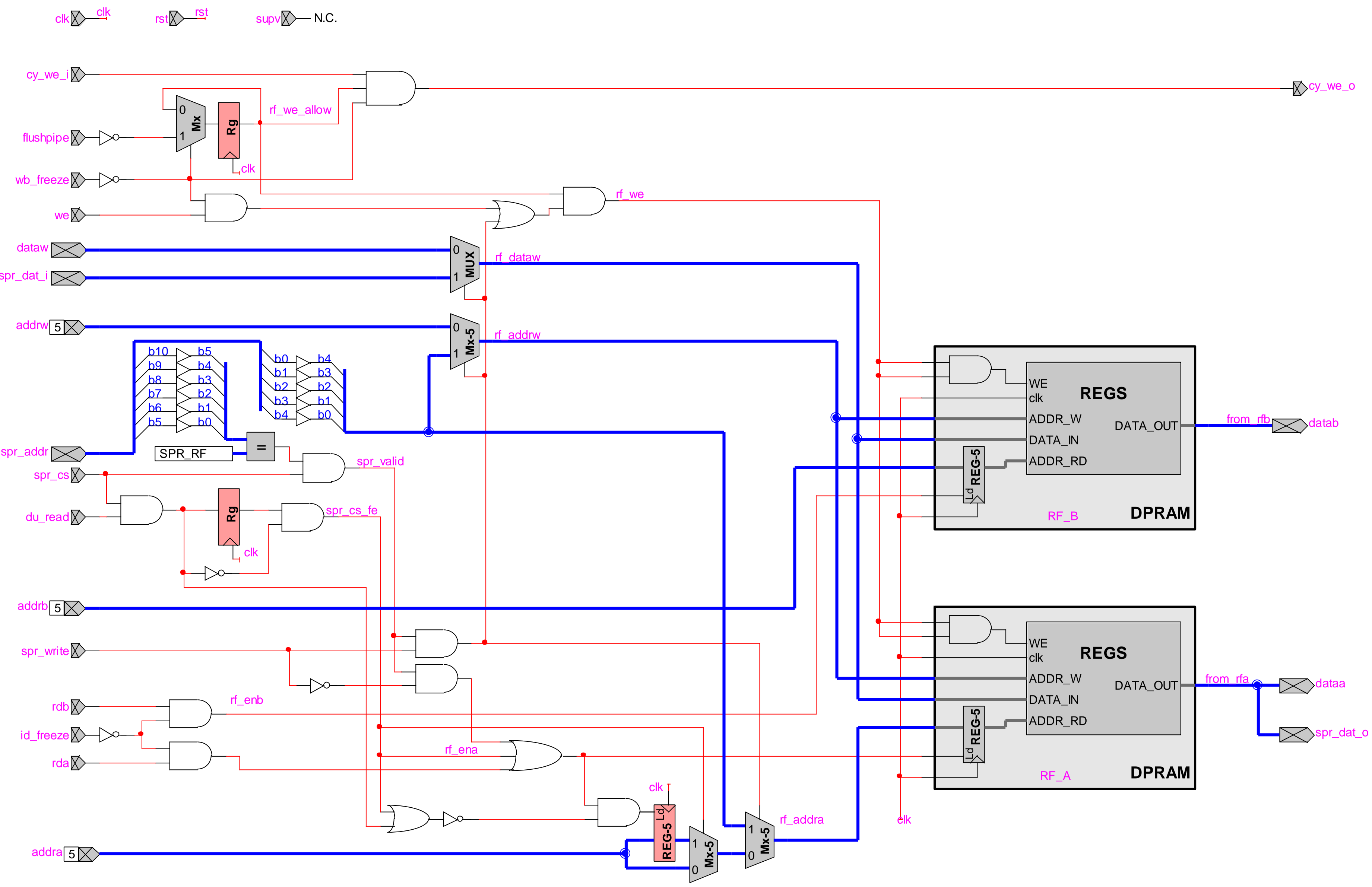


Il·lustració C-1. Circuit lògic mòdul OR1200's Generate PC

OR1200's Instruction Fetcher



Il·lustració C-2. Circuit lògic mòdul OR1200's Instruction Fetcher

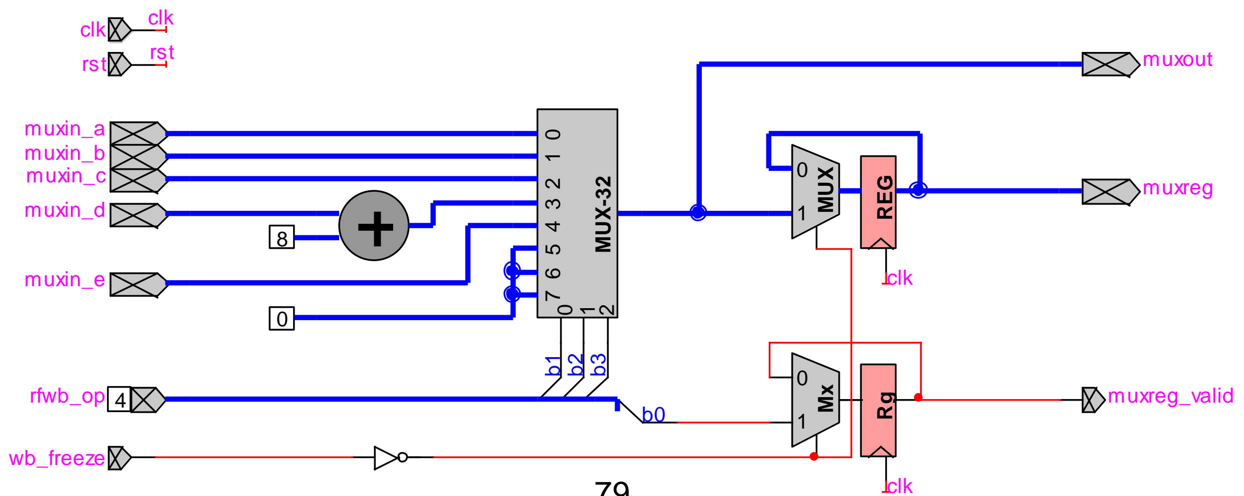


Il·lustració C-3. Circuit lògic mòdul OR1200's Register File

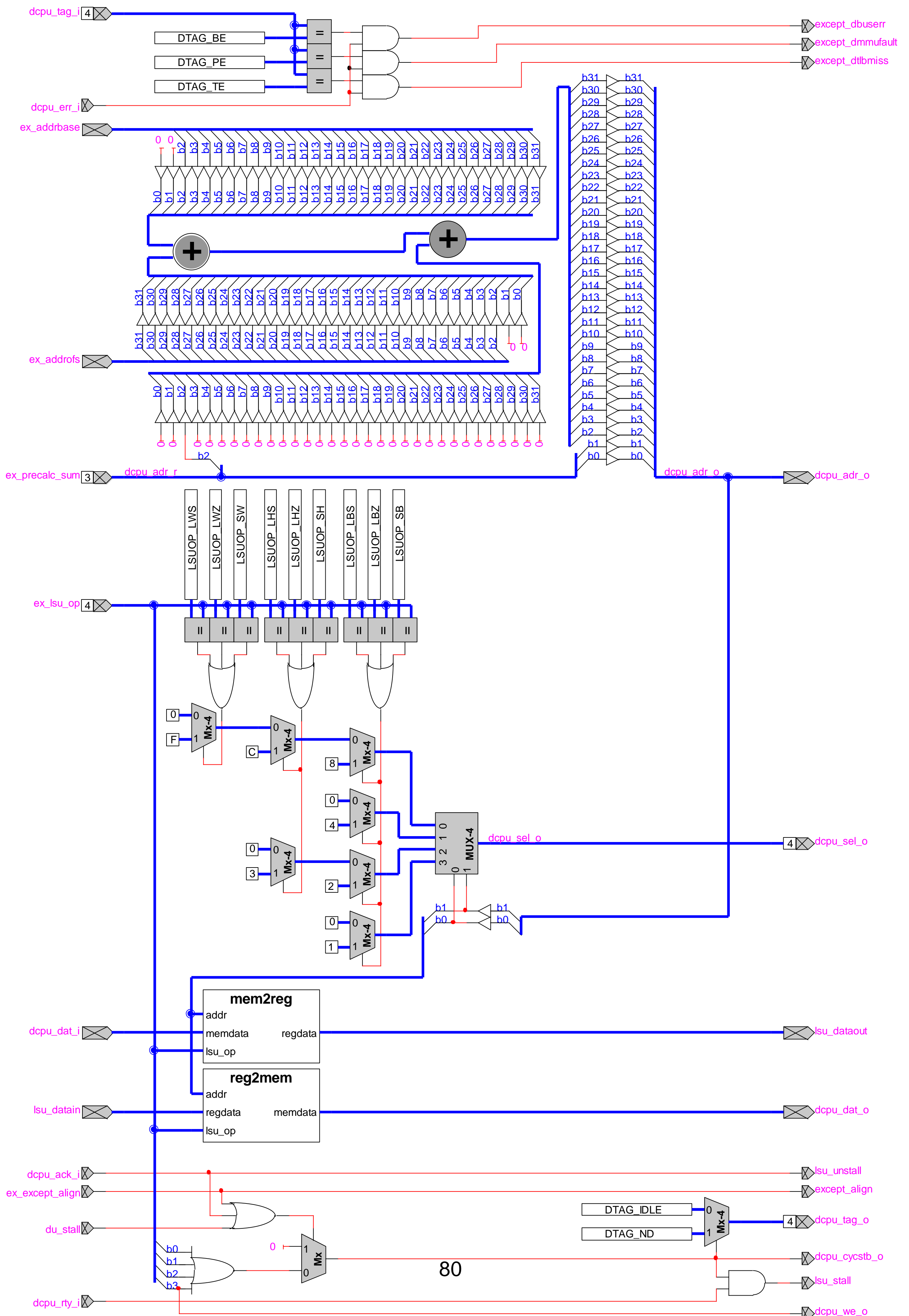
OR1200's register file read operands mux

Il·lustració C-4. Circuit lògic mòdul OR1200's register file read operands Mux

OR1200's Write-back Mux

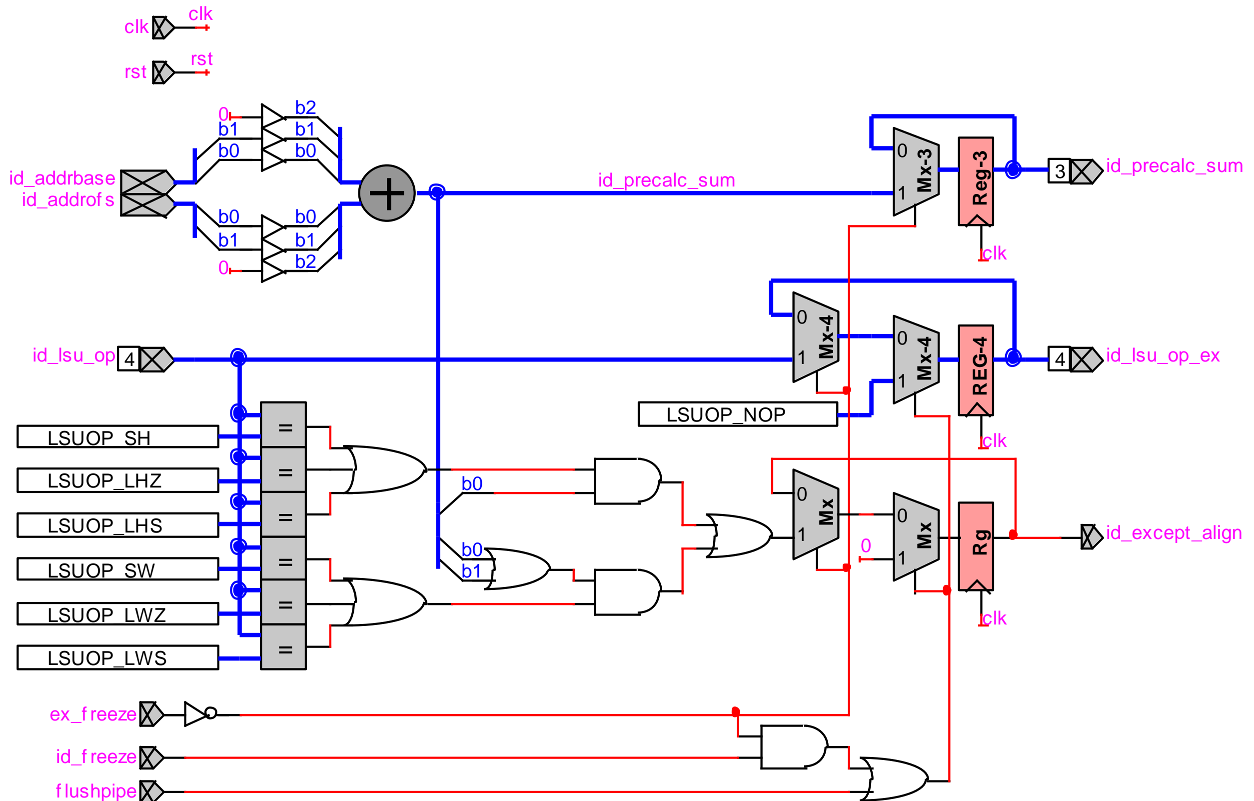


Il·lustració C-5. Circuit lògic mòdul OR1200's Write-back Mux



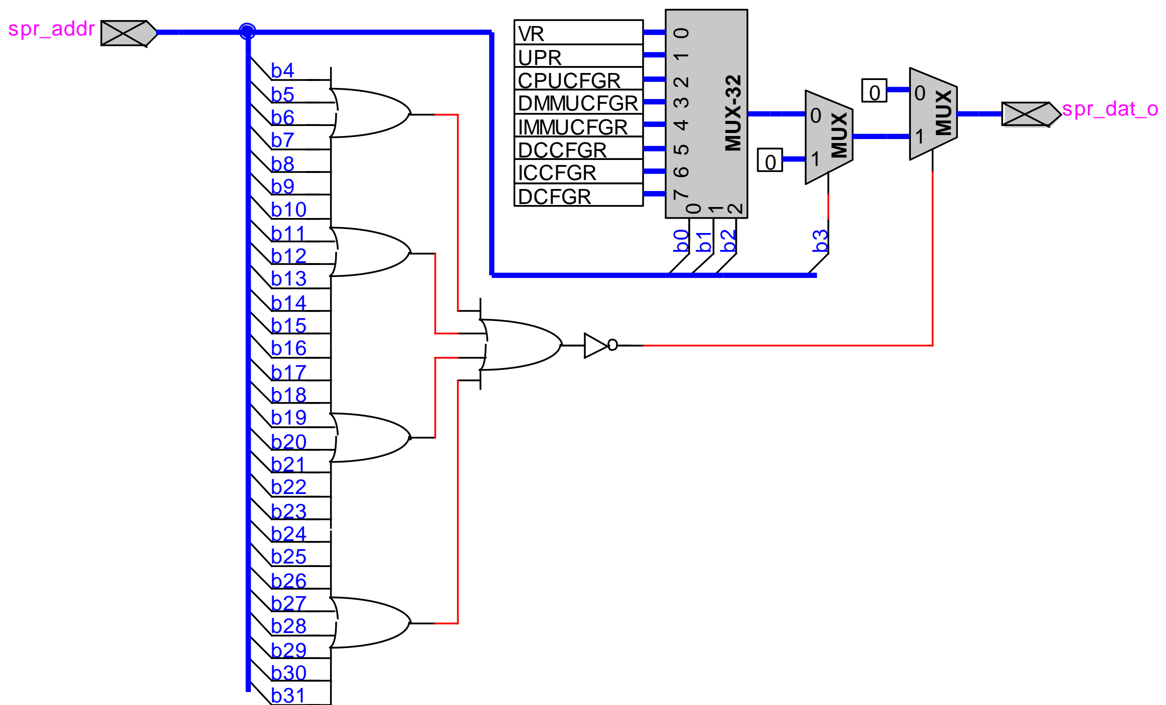
Il·lustració C-6. Circuit lògic mòdul OR1200's Load / Store Unit (EX stage)

OR1200's Load / Store Unit (ID stage)



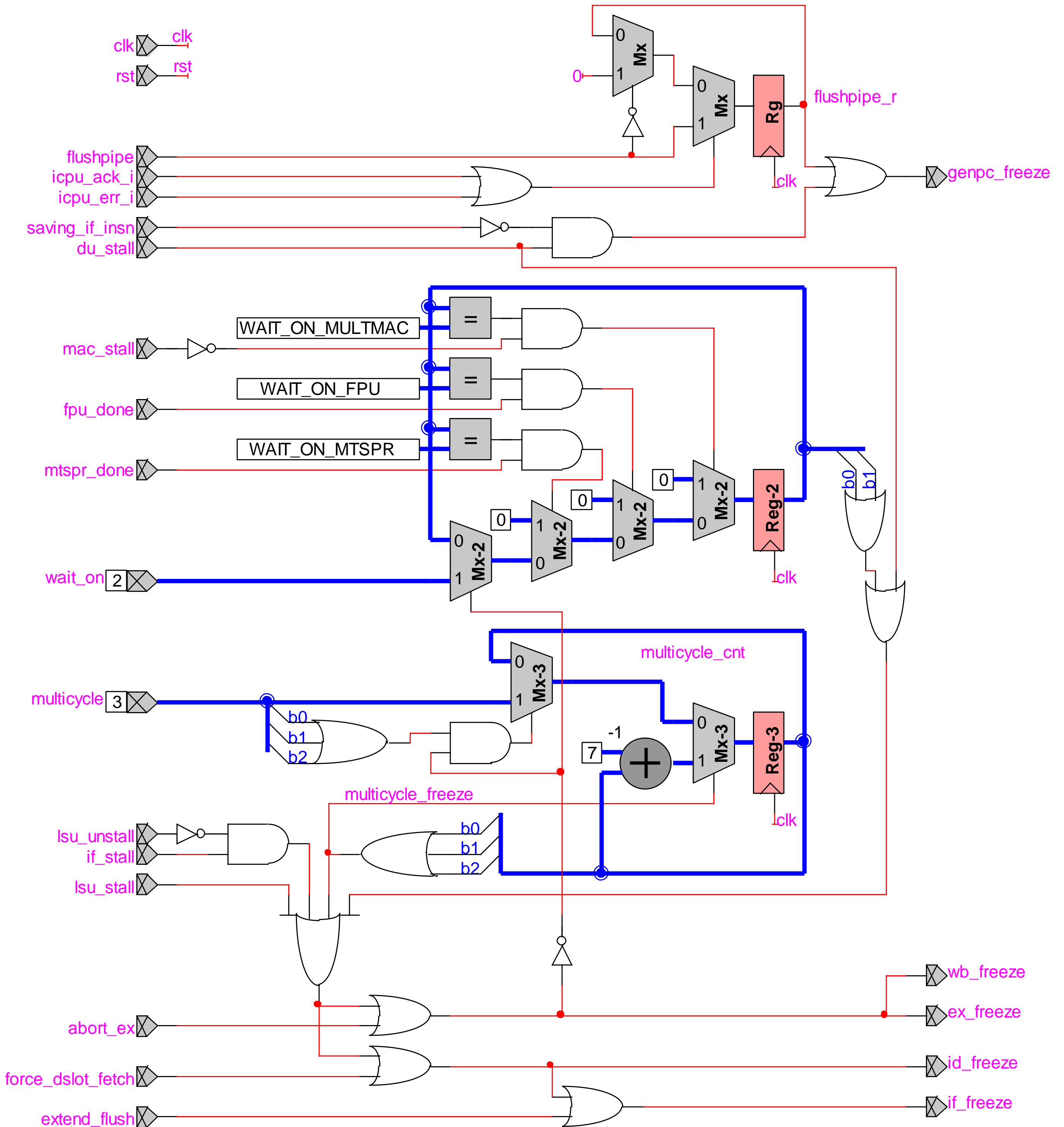
Il·lustració C-7. Circuit lògic mòdul OR1200's Load / Store Unit (ID stage)

OR1200's VR, UPR and Configuration Registers



Il·lustració C-8. Circuit lògic mòdul OR1200's VR, UPR and Configuration Registers (abans de PCU)

OR1200's Freeze logic



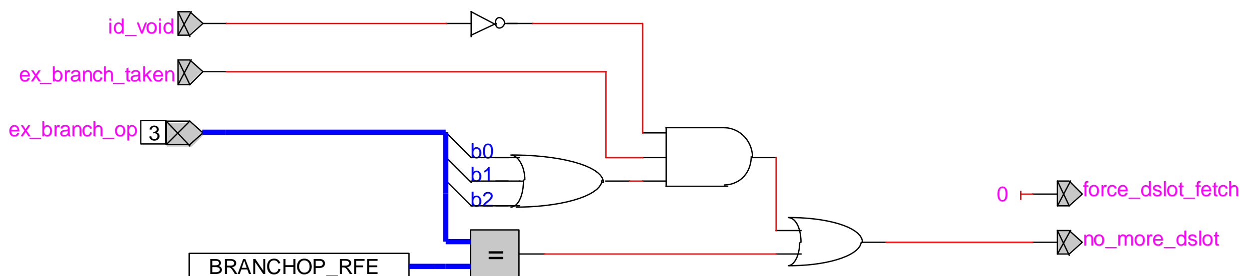
Il·lustració C-9. Circuit lògic mòdul OR1200's Freeze logic

OR1200's Instruction decode IF stage

if_insn
clk
rst
b20 b19 b18 b17 b16
b4 b3 b2 b1 b0
5 rf_addra
b15 b14 b13 b12 b11
b4 b3 b2 b1 b0
5 rf_addrb
b31
MAC is not implemented
0
rf_rda
b30
rf_rdb
MUX
MUX
NOP
REG
id_insn
clk
OR32_RFE OR32_BF OR32_BNF OR32_JR OR32_JALR OR32_JAL OR32_J
b31 b30 b29 b28 b27 b26
BRANCHOP_J
BRANCHOP_JR
BRANCHOP_BNF
BRANCHOP_BF
BRANCHOP_RFE
BRANCHOP_NOP
Mx-3
Reg-3
3 id_branch_op
BRANCHOP_NOP
id_freeze
except_flushpipe
pc_we
extend_flush
du_flush_pipe
id_flushpipe
if_flushpipe

Il·lustració C-10. Circuit lògic mòdul OR1200's Instruction decode (IF stage)

no_more_dslot (ID) CTRL



Il·lustració C-11. Circuit lògic senyal 'no_more_dslot' mòdul OR1200's Instruction decode (ID stage)

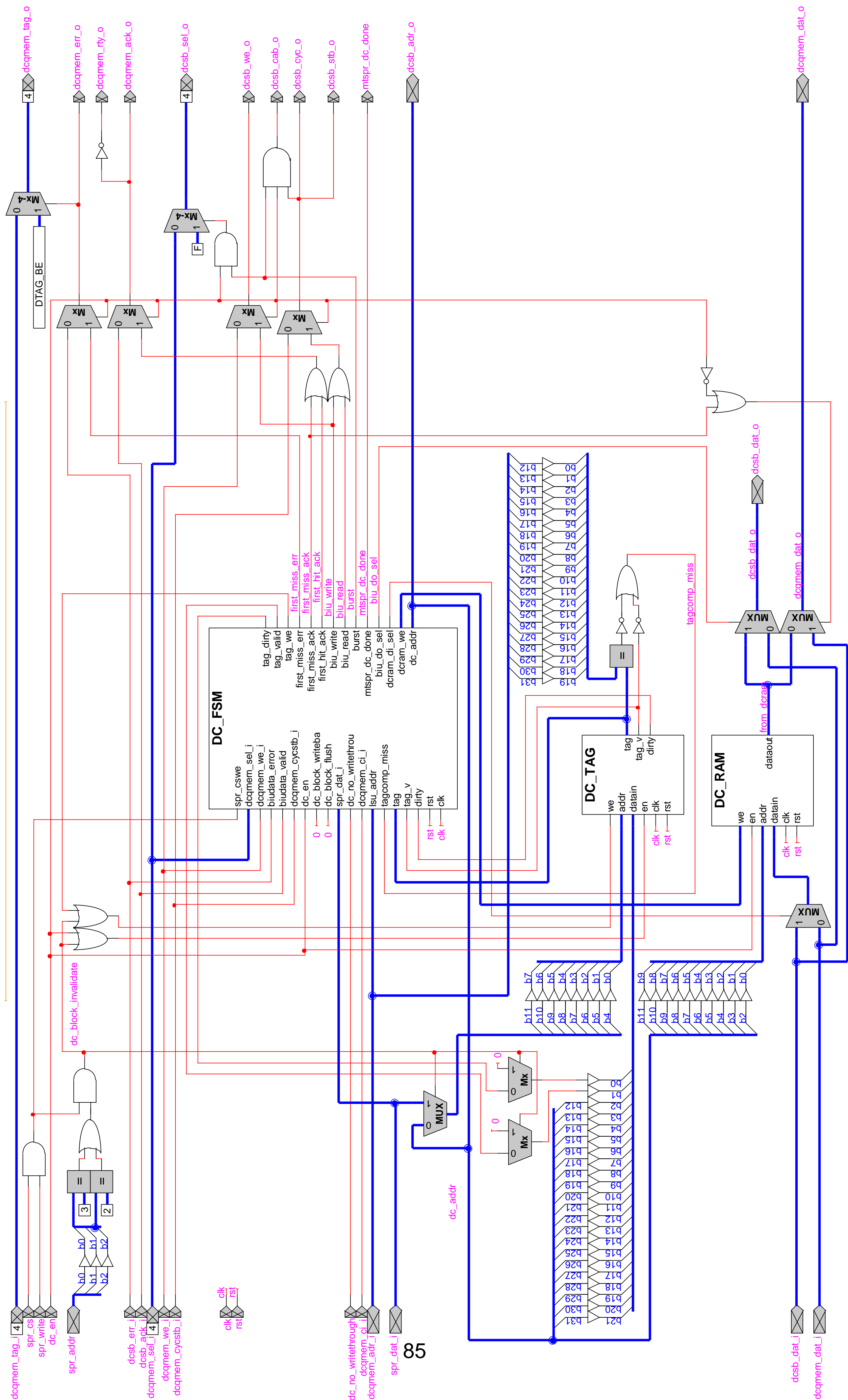
OR1200's Instruction decode EX stage

clk clk
rst rst
except_flushpipe
pc_we
extend_flush
du_flush_pipe
ex_flushpipe
wb_flushpipe
ex_flushpipe
ex_insn
b31 b30 b29 b28 b27 b26
b5 b4 b3 b2 b1 b0
=
OR32_NOP
b16
ex_void
b4 b3 b2 b1 b0
b4 b3 b2 b1 b0
5 cust5_op
b5 b6 b7 b8 b9 b10
b5 b4 b3 b2 b1 b0
6 cust5_limm
0 MUX 1
REG
wb_insn
clk
0 Mx-5 1
REG-5
ex_rfaddrw 5
5 wb_rfaddrw
wb_freeze
clk

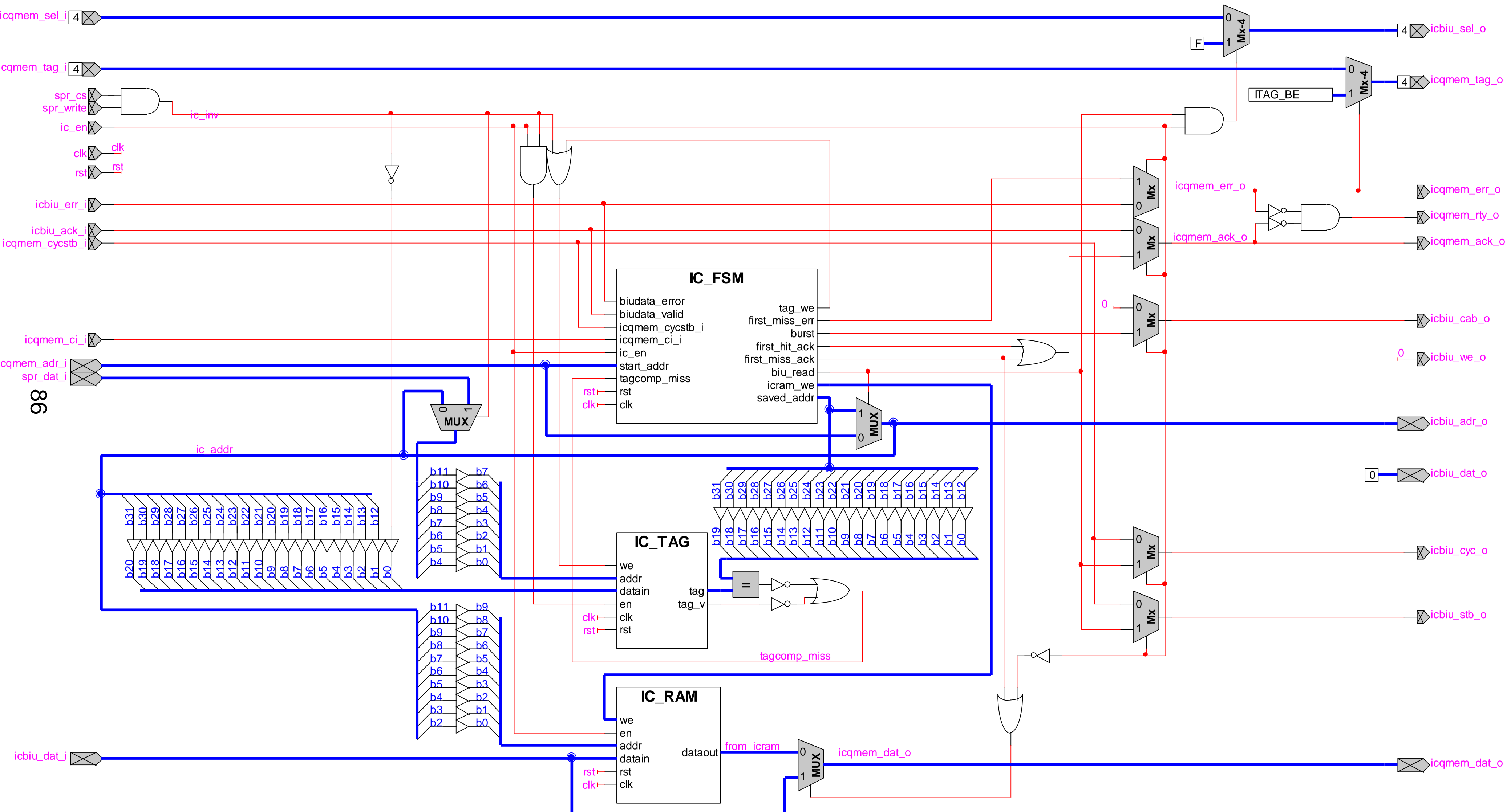
Il·lustració C-12. Circuit lògic mòdul OR1200's Instruction decode (EX stage)

OR1200's Instruction decode WB stage

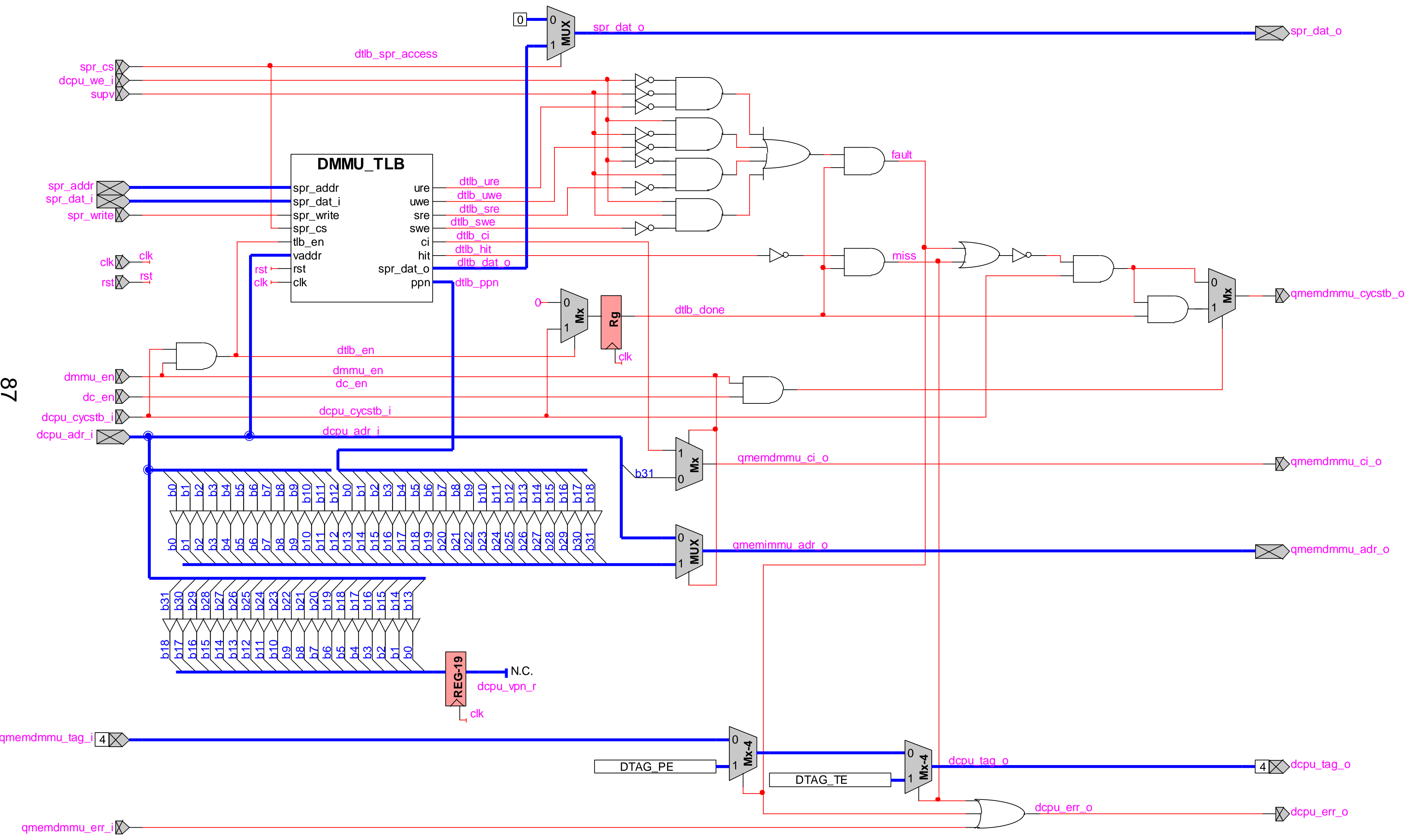
Il·lustració C-13. Circuit lògic mòdul OR1200's Instruction decode (WB stage)



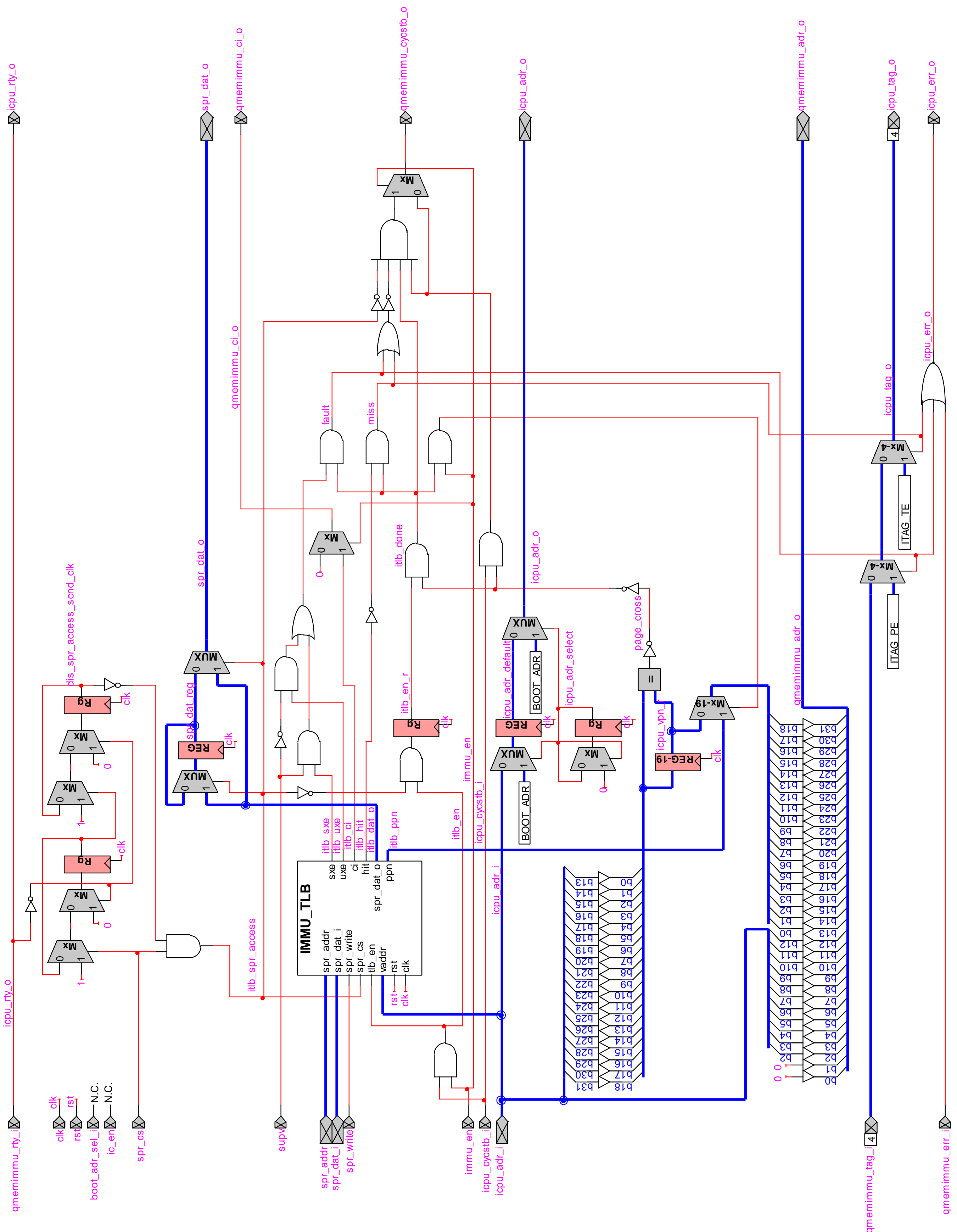
Il·lustració C-14. Circuit lògic mòdul OR1200's Data Cache Top Level



Il·lustració C-15. Circuit lògic mòdul OR1200's Instruction Cache Top Level



Il·lustració C-16. Circuit lògic mòdul OR1200's Data MMU top level



Il·lustració C-17. Circuit lògic mòdul OR1200's Instruction MMU top level

ANNEX D JOC DE PROVES: PCU

El joc de proves comprova cada esdeveniment (exceptuant els WPE) i indica el resultat esperat justificat. Els comentaris s'han fet en anglès perquè així el joc de proves pugui ser integrat dins del projecte OpenRISC.

D.1 Codi del joc de proves:

```
1  /* PCU tests */
2  #include "spr-defs.h"
3
4  // ================================================== [ exceptions ] ===
5        .section .vectors, "ax"
6
7  // ---[ 0x100: RESET exception ]----------------------------------------
8          .org 0x100
9        l.movhi r0, 0
10       l.movhi r1, 0
11       l.movhi r2, 0
12       l.movhi r3, 0
13       l.movhi r4, 0
14       l.movhi r5, 0
15       l.movhi r6, 0
16       l.movhi r7, 0
17       l.movhi r8, 0
18       l.movhi r9, 0
19       l.movhi r10, 0
20       l.movhi r11, 0
21       l.movhi r12, 0
22       l.movhi r13, 0
23       l.movhi r14, 0
24       l.movhi r15, 0
25       l.movhi r16, 0
26       l.movhi r17, 0
27       l.movhi r18, 0
28       l.movhi r19, 0
29       l.movhi r20, 0
30       l.movhi r21, 0
31       l.movhi r22, 0
32       l.movhi r23, 0
33       l.movhi r24, 0
34       l.movhi r25, 0
35       l.movhi r26, 0
36       l.movhi r27, 0
37       l.movhi r28, 0
38       l.movhi r29, 0
39       l.movhi r30, 0
40       l.movhi r31, 0
41
42       // Jump to program initialization code
43       .global _start
44       l.movhi r4, hi(_start)
45       l.ori r4, r4, lo(_start)
46       l.jr    r4
47       l.nop
```

```
48
49  // ---[ 0x900: dtlb miss ]----------------------------------------------
50          .org 0x900
51          // DTLB miss exception handler
52
53          // Check first 'DTLB Miss Event Test' below to figure out why we use
54          // these constants and values
55
56          // Get EA that cause the exception
57          l.mfspr     r6, r0, SPR_EEAR_BASE   // r6 = EA
58
59          // Find TLB set
60          l.srli      r4, r6, 0xD       // (EA / pageSize) % numSets; numSets=64
61          l.andi      r4, r4, 0x3F      // r4 = set
62
63          // Whatever access is in progress, translated address have to point to
64          // physical RAM. According to below params, RAM addr test starts at 4MB.
65
66          l.movhi     r2, 0x0040
67          l.slli      r3, r4, 0xD       // set*pageSize
68          l.add r3, r2, r3        // r3 = TA = ram_addr + set*pageSize
69
70          l.movhi     r7, hi(SPR_DTLBMR_VPN)
71          l.ori r7, r7, lo(SPR_DTLBMR_VPN)
72          l.and r6, r6, r7
73          l.movhi     r7, hi(SPR_DTLBTR_PPN)
74          l.ori r7, r7, lo(SPR_DTLBTR_PPN)
75          l.and r3, r3, r7
76
77          l.ori r6, r6, SPR_DTLBMR_V    // ea | SPR_DTLBMR_V
78          l.ori r3, r3, DTLB_PR_NOLIMIT // ta | DTL_PR_NOLIMIT
79
80          // Set DTLB entry
81          l.mtspr     r4, r6, SPR_DTLBMR_BASE(0)
82          l.mtspr     r4, r3, SPR_DTLBTR_BASE(0)
83
84          l.rfe
85
86  // ---[ 0xa00: itlb miss ]----------------------------------------------
87          .org 0xa00
88          // ITLB miss exception handler
89
90          // Check first 'ITLB Miss Event Test' below to figure out why we use
91          // these constants and values
92
93          // Get EA that cause the exception
94          l.mfspr     r6, r0, SPR_EEAR_BASE   // r6 = EA
95
96          // Find TLB set
97          l.srli      r4, r6, 0xD       // (EA / pageSize) % numSets; numSets=64
98          l.andi      r4, r4, 0x3F      // r4 = set
99
100         // Whatever access is in progress, translated address have to point to
101         // physical RAM. According to below params, RAM addr test starts at 4MB.
102
103         l.movhi     r2, 0x0040
104         l.slli      r3, r4, 0xD       // set*pageSize
105         l.add r3, r2, r3        // r3 = TA = ram_addr + set*pageSize
106
107         l.movhi     r7, hi(SPR_ITLBMR_VPN)
108         l.ori r7, r7, lo(SPR_ITLBMR_VPN)
109         l.and r6, r6, r7
110         l.movhi     r7, hi(SPR_ITLBTR_PPN)
```

```
111         l.ori r7, r7, lo(SPR_ITLBTR_PPN)
112         l.and r3, r3, r7
113 
114         l.ori r6, r6, SPR_ITLBMR_V    // ea | SPR_ITLBMR_V
115         l.ori r3, r3, ITLB_PR_NOLIMIT // ta | ITL_PR_NOLIMIT
116 
117         // Set ITLB entry
118         l.mtspr     r4, r6, SPR_ITLBMR_BASE(0)
119         l.mtspr     r4, r3, SPR_ITLBTR_BASE(0)
120 
121         l.rfe
122 
123   // ---[ 0xc00: sys call ]-----------------------------------------------
124           .org 0xc00
125         // System call handler
126 
127         // Set Supervisor mode
128         l.mfspr     r6, r0, SPR_ESR_BASE
129         l.ori r6, r6, SPR_SR_SM
130         l.mtspr     r0, r6, SPR_ESR_BASE
131 
132         l.rfe
133 
134   // ==================================================== [    text    ] ===
135         .section .text
136 
137         .global _start
138   _start:
139         l.addi      r1, r0, 1
140         l.jal   _main     // Kick off program commenting this line
141         l.nop
142 
143   /////////////////////////////////////////////////////////////////////////
144   /////////////////////////////////////////////////////////////////////////
145   /////////////////////////////////////////////////////////////////////////
146 
147   // ==================================================== [    main    ] ===
148         .global _main
149   _main:
150         ////////////////////////////////////////////////////////////////////
151         // Select which tests do (1st tests or 2nd tests) because there
152         // are more tests than couters. Comment this line to execute 1st tests.
153         // Uncomment it to execute 2nd tests:
154 
155         //    l.addi      r1, r0, 2
156         //
157         ////////////////////////////////////////////////////////////////////
158 
159         l.sfeqi     r1, 0x2
160         l.bf   _secondsTests
161         l.nop
162 
163     ///////////////////////////////////////////////////////////////////////
164     // Firsts tests
165 
166     _firstsTests:
167 
168         // Only one counter and event is active at the same time
169 
170               l.jal _IF_test    // R20 - PCCR0 - SW0    // Result: 0x14de1
171               l.nop
172               l.jal _LA_test    // R21 - PCCR1 - SW1    // Result: 0x50
173               l.nop
```

```
174             l.jal _SA_test    // R22 - PCCR2 - SW2    // Result: 0x8c
175             l.nop
176             l.jal _DCM_test   // R23 - PCCR3 - SW3    // Result: 0xdc
177             l.nop
178             l.jal _ICM_test   // R24 - PCCR4 - SW4    // Result: 0x4
179             l.nop
180             l.jal _IFS_test   // R25 - PCCR5 - SW5    // Result: 0x117 - 0x118
181             l.nop
182             l.jal _BS_test    // R26 - PCCR6 - SW6    // Result: 0x11
183             l.nop
184             l.jal _LSUS_test  // R27 - PCCR7 - SW7    // Result: 0x124 - others
185             l.nop
186
187             l.j   _finishPROG
188             l.nop
189   //
190   ////////////////////////////////////////////////////////////////////////
191
192   ////////////////////////////////////////////////////////////////////////
193   // Seconds tests
194
195   _secondsTests:
196
197       // Only one counter and event is active at the same time
198
199             l.jal _DTLBM_test // R28 - PCCR0 - SW0    // Result: 0x4
200             l.nop
201             l.jal _ITLBM_test // R29 - PCCR1 - SW1    // Result: 0x3
202             l.nop
203
204             // Note: No Hardware watchpoints test is provided due they
205             // aren't implemented in this board
206
207       //-------------------------------------------------------------------
208
209       // Count more than one event in same counter. Check privilege mode to
210       // count. Useful MEM access: Load + Store accesses
211
212             l.jal _SALA_test  // R30 - PCCR2 - SW2    // Result: 0x6
213             l.nop
214
215       //-------------------------------------------------------------------
216
217       // More than one counter is active at the same time
218
219             l.jal _combined_test
220             l.nop                   // IF  - PCCR3 - SW3    // Result: 0x1b
221                                     // ICM - PCCR4 - SW4    // Result: 0x6 - 0x7
222                                     // LA  - PCCR5 - SW5    // Result: 0x1
223                                     // BS  - PCCR6 - SW6    // Result: 0x3
224                                     // SA  - PCCR7 - SW7    // Result: 0x1
225   //
226   //////////////////////////////////////////////////////////
227
228   _finishPROG:
229       // Finish program
230       l.j   0
231       l.nop 0x1   // Exit simulation
232
233 /////////////////////////////////////////////////////////////////
234 /////////////////////////////////////////////////////////////////
235
236 /////////////////////////////////////////////////////////////////
```

```
237 // Instruction Fetch Event test in PCCR0
238 //
239 _IF_test:
240       // Initialization
241
242       l.mtspr     r0, r0, SPR_PCCR(0)
243       l.addi      r3, r0, (SPR_PCMR_CISM | SPR_PCMR_CIUM) // Count in SM & in UM
244       l.ori r3, r3, SPR_PCMR_IF
245       l.mtspr     r0, r3, SPR_PCMR(0)
246       // Start Counting //
247
248       l.addi      r3, r0, 0x2fb2 // 12210
249
250   _ifLOOP:
251       l.addi      r4, r0, 0x2
252       l.addi      r5, r0, 0x3
253       l.xor r6, r4, r5
254       l.addi      r3, r3, -1
255       l.sfeqi     r3, 0
256       l.bnf _ifLOOP
257       l.or  r6, r3, r4 // DS
258
259       l.and r5, r3, r4
260
261       // Finish Counting //
262       l.mtspr     r0, r0, SPR_PCMR(0)  // Fetch already done when counter stops
263
264       // Result = 1(addi) + 7*12210(bucle) + 1(and) + 1(mtspr) =
265       //             = 85470 = 0x14de1
266
267       l.mfspr     r20, r0, SPR_PCCR(0) // Read counter to GPRs
268
269       l.jr  r9 // return
270       l.nop
271 //
272 /////////////////////////////////////////////////////////////////
273
274 /////////////////////////////////////////////////////////////////
275 // Load Access Event test
276 //
277 _LA_test:
278       // Initialization
279       l.movhi     r31, 0x0000
280       l.ori r31, r31, 0x0040
281       l.movhi r5, 0x1234
282     l.ori   r5, r5, 0x5678
283
284     l.sw    0(r31), r5 // Putting some value in memory
285
286       l.mtspr     r0, r0, SPR_PCCR(1)
287       l.addi      r3, r0, (SPR_PCMR_CISM | SPR_PCMR_CIUM)
288       l.ori r3, r3, SPR_PCMR_LA
289       l.mtspr     r0, r3, SPR_PCMR(1)
290       // Start Counting //
291
292       l.addi      r3, r0, 0xA // 10
293
294   _laLOOP:  // big endian
295       l.lwz r4, 0(r31)  // 1st
296       l.sfne      r4, r5
297       l.bf  _laLOOPend
298       l.addi      r4, r0, 0x0
299
```

```
300         l.lhz r4, 0(r31)  // 2nd
301         l.lhz r6, 2(r31)  // 3rd
302         l.slli      r4, r4, 0x10
303         l.or  r4, r4, r6
304         l.sfne      r4, r5
305         l.bf  _laLOOPend
306         l.addi      r4, r0, 0x0
307
308         l.lbz r4,   0(r31)      // 4th
309         l.slli      r4, r4, 0x18      // [31:24]
310         l.lbz r6,   1(r31)      // 5th
311         l.slli      r6, r6, 0x10      // [23:16]
312         l.or  r4, r4, r6
313         l.lbz r6,   2(r31)      // 6th
314         l.slli      r6, r6, 0x8       // [15:8]
315         l.or  r4, r4, r6
316         l.lbz r6,   3(r31)      // 7th
317         l.or  r4, r4, r6
318         l.sfne      r4, r5
319         l.bf  _laLOOPend
320         l.addi      r4, r0, 0x0
321
322         l.sw  0(r31), r5
323
324         l.addi      r3, r3, -1
325         l.sfeqi     r3, 0
326         l.bnf _laLOOP
327         l.lwz r7, 0(r31) // 8th
328
329     _laLOOPend:
330
331         // Finish Counting //
332         l.mtspr     r0, r0, SPR_PCMR(1)
333
334         // Result = 8(loads)*10 = 80 = 0x50 (if not, there is an error)
335
336         l.mfspr     r21, r0, SPR_PCCR(1) // Read counter to GPRs
337
338         l.jr  r9 //return
339         l.nop
340   //
341   /////////////////////////////////////////////////////////////////
342
343   /////////////////////////////////////////////////////////////////
344   // Store Access Event test
345   //
346   _SA_test:
347         // Initialization
348         l.movhi     r31, 0x0000
349         l.ori r31, r31, 0x0040
350
351         l.mtspr     r0, r0, SPR_PCCR(2)
352         l.addi      r3, r0, (SPR_PCMR_CISM | SPR_PCMR_CIUM)
353         l.ori r3, r3, SPR_PCMR_SA
354         l.mtspr     r0, r3, SPR_PCMR(2)
355         // Start Counting //
356
357         l.addi      r3, r0, 0x14 // 20
358
359     _saLOOP:  // big endian
360       l.addi  r4, r0, 0x12
361       l.addi  r5, r0, 0x34
362       l.addi  r6, r0, 0x56
```

```
363         l.addi  r7, r0, 0x78
364         l.sb    0(r31), r4  // 1st
365         l.sb    1(r31), r5  // 2nd
366         l.sb    2(r31), r6  // 3rd
367         l.sb    3(r31), r7  // 4th
368
369         l.addi  r4, r0, 0x1234
370         l.addi  r5, r0, 0x5678
371         l.sh    4(r31), r4  // 5th
372         l.sh    6(r31), r5  // 6th
373
374         l.slli  r4, r4, 0x10
375         l.or    r4, r4, r5
376         l.sw    8(r31), r4 // 7st
377
378         l.lwz   r5, 0(r31)
379         l.lwz   r6, 4(r31)
380         l.lwz   r7,   8(r31)
381
382         l.and   r5, r5, r6
383         l.and   r5, r5, r7
384         l.sfne  r5, r4
385         l.bf    _saLOOPend
386
387           l.addi      r3, r3, -1
388           l.sfeqi     r3, 0
389           l.bnf _saLOOP
390           l.nop
391
392       _saLOOPend:
393
394           // Finish Counting //
395           l.mtspr     r0, r0, SPR_PCMR(2)
396
397           // Result = 7(stores)*20 = 140 = 0x8C (if not, there is an error)
398
399           l.mfspr     r22, r0, SPR_PCCR(2) // Read counter to GPRs
400
401           l.jr  r9 //return
402           l.nop
403     //
404     ////////////////////////////////////////////////////////////////
405
406     ////////////////////////////////////////////////////////////////
407     // Data Cache Miss Event test
408     //
409     _DCM_test:
410           // Initialization
411
412           // DC of 4KB, 256 sets, 16 bytes/set
413           // Write-through policy
414
415           l.addi      r14, r0, 0x10     // 16, Cache block size
416           l.addi  r5, r0, 0x8           // 8, log2(# of cache sets)
417           l.addi   r7, r0, 0x100  // 256, Number of cache sets
418
419         // Invalidate DC
420         l.addi  r6,r0,0
421         l.sll   r5,r14,r5
422       .invDCM_Loop:
423         l.mtspr r0,r6,SPR_DCBIR
424         l.sfne  r6,r5
425         l.bf    .invDCM_Loop
```

```
426          l.add   r6,r6,r14
427
428        // Enable DC
429        l.mfspr r6,r0,SPR_SR
430        l.ori   r6,r6,SPR_SR_DCE
431        l.mtspr r0,r6,SPR_SR
432
433          l.mtspr     r0, r0, SPR_PCCR(3)
434          l.addi      r3, r0, (SPR_PCMR_CISM | SPR_PCMR_CIUM)
435          l.ori r3, r3, SPR_PCMR_DCM
436          l.mtspr     r0, r3, SPR_PCMR(3)
437          // Start Counting //
438
439          l.addi      r31, r0, 0x5000   // Address
440          l.addi      r3, r0, 0xDC      // 220, iteration
441          l.movhi     r5,    0x5ABC           // data
442          l.ori r5, r5, 0xDEF5
443
444      _dcmLOOP:                   // 1st   |  2nd --> (cyclic every 2 its.)
445          l.sw  0(r31),     r5    //  M         H
446          l.lwz r4,    0(r31)      //  M         H
447          l.lhz r6,    2(r31)      //  H         H
448          l.sw  4(r31), r5         //  H         H
449          l.addi      r3, r3, -1
450          l.sfeqi     r3, 0
451          l.bnf _dcmLOOP
452          l.addi      r31, r31, 0x8     // DS
453
454          // Finish Counting //
455          l.mtspr     r0, r0, SPR_PCMR(3)
456
457        // Disable DC
458        l.mfspr r6, r0, SPR_SR
459        l.addi  r5, r0, -1
460        l.xori  r5, r5, SPR_SR_DCE
461          l.and r5, r6, r5
462        l.mtspr r0, r5, SPR_SR
463
464          // Result = 2 Miss * 220/2 (it) = 220 = 0xDC
465
466          l.mfspr     r23, r0, SPR_PCCR(3) // Read counter to GPRs
467
468          l.jr  r9 //return
469          l.nop
470    //
471    ////////////////////////////////////////////////////////////////
472
473    ////////////////////////////////////////////////////////////////
474    // Instruction Cache Miss Event test
475    //
476    _ICM_test:
477          // Initialization
478
479          // IC of 4KB, 256 sets, 16 bytes/set
480          // Instructions of 4 bytes.
481          // 4 instr/set --> 1 miss every 4 instructions
482
483          l.addi      r14, r0, 0x10     // 16, Cache block size
484          l.addi  r5, r0, 0x8           // 8, log2(# of cache sets)
485          l.addi   r7, r0, 0x100  // 256, Number of cache sets
486
487          // Invalidate IC
488          l.addi      r6, r0, 0x0
```

```
489         l.sll   r5, r14, r5
490     .invICM_Loop:
491         l.mtspr     r0, r6, SPR_ICBIR
492         l.sfne      r6, r5
493         l.bf  .invICM_Loop
494         l.add r6, r6, r14
495
496         // Enable IC
497         l.mfspr r6, r0, SPR_SR
498         l.ori   r6, r6, SPR_SR_ICE
499         l.mtspr r0, r6, SPR_SR
500         l.nop
501         l.nop
502         l.nop
503         l.nop
504         l.nop
505         l.nop
506         l.nop
507         l.nop
508
509         l.mtspr     r0, r0, SPR_PCCR(4)
510         l.addi      r3, r0, (SPR_PCMR_CISM | SPR_PCMR_CIUM)
511         l.ori r3, r3, SPR_PCMR_ICM
512         l.mtspr     r0, r3, SPR_PCMR(4)
513         // Start Counting //
514
515         l.addi      r4,   r0,   0x4   // Group of 4 instr, 1 miss
516         l.addi      r5, r4, 0x1
517         l.addi      r6, r4, 0x2
518         l.addi      r3,   r0, 0xA
519
520     _icmLOOP: // 10 times loop. Only 1 miss.
521         l.addi      r3, r3, -1
522         l.sfeqi     r3, 0
523         l.bnf _icmLOOP
524         l.and r7, r4, r5
525
526         l.ori r5, r5, 0xF // Group of 4 instr, 1 miss
527         l.xor r6, r4, r5
528         l.movhi     r3,   0x3333
529         l.and r2, r2, r2
530
531         l.and r2, r2, r2  // Group of 4 instr (including mtspr), 1 miss
532         l.and r2, r2, r2
533         l.and r2, r2, r2
534
535         // Finish Counting //
536         l.mtspr     r0, r0, SPR_PCMR(4)  // Fetch already done when counter stops
537
538         // Disable IC
539         l.mfspr r6, r0, SPR_SR
540         l.addi  r5, r0, -1
541         l.xori  r5, r5, SPR_SR_ICE
542         l.and   r5, r6, r5
543         l.mtspr r0, r5, SPR_SR
544
545         // Result = 1 + 1 (loop, only 1) + 1 + 1 = 4 = 0x4
546
547         l.mfspr     r24, r0, SPR_PCCR(4) // Read counter to GPRs
548
549         l.jr  r9 //return
550         l.nop
551   //
```

```
552  ////////////////////////////////////////////////////////////////
553
554  ////////////////////////////////////////////////////////////////
555  // Instruction Fetch Stall Event test
556  //
557  _IFS_test:
558        // Initialization
559
560        l.mtspr     r0, r0, SPR_PCCR(5)
561        l.addi      r3, r0, (SPR_PCMR_CISM | SPR_PCMR_CIUM)
562        l.ori r3, r3, SPR_PCMR_IFS
563        l.mtspr     r0, r3, SPR_PCMR(5)     // This instruction is also counted
564        // Start Counting //
565
566        // The memory controller stalls IF 8 cycles when search instructions
567        // to main memory and then, sents 1 instruction through Wishbone bus
568        // every 4 cycles (during 3 cycles IF is stalled). This configuration
569        // is without Instruction Cache.
570
571        // Some instructions
572        l.addi      r4, r0, 0x4
573        l.addi      r5, r0, 0x5
574        l.addi      r6, r0, 0x6
575        l.addi      r7, r0, 0x7
576        l.addi      r8, r0, 0x8
577        l.and r5, r4, r5
578        l.addi      r3, r0, 0x6
579
580    _ifsLOOP:
581        l.or  r6, r5, r6
582        l.or  r7, r6, r7
583        l.xor r4, r6, r7
584        l.andi      r2, r4, 0xF
585        l.addi      r3, r3, -1
586        l.sfeqi     r3, 0
587        l.bnf _ifsLOOP
588        l.srai      r5, r8, 0x1 // DS
589
590        // Finish Counting //
591        l.mtspr     r0, r0, SPR_PCMR(5)
592
593        // When the jump or branch is in EX stage and its taken, the new
594        // instruction will be available after 2 cycles. In the first one,
595        // there is a valid instruction in the delay slot. In the second
596        // one, there aren't more valid instructions and a nop is inserted.
597        // So every time that a branch or jump is taken, a nop is inserted
598        // into the pipeline.
599
600
601        // Result = 9(8cyc + 3cyc*8insn)*(1 + 6 its)) +
602        //          + 8cyc * 5brTaken + 3cyc * 5 nops [ +- 1 cycle] = 279 = 0x117
603        //                                                               0x118
604
605        // NOTE: The result can be different to 0x117 depending on the memory.
606        // We have an SDRAM, and there are refreshing cycles or the necessity of
607        // charging current row. Particularly, will be 0x117 if this test is run
608        // alone and 0x118 with all tests enabled. Use a signal / wave viewer to
609        // check its correctness. There are not more cycles because the code
610        // which is being counted is really small.
611
612        l.mfspr     r25, r0, SPR_PCCR(5) // Read counter to GPRs
613
614        l.jr  r9 //return
```

```
615         l.nop
616   //
617   ////////////////////////////////////////////////////////////////
618
619   ////////////////////////////////////////////////////////////////
620   // Branch Stall Event test
621   //
622   _BS_test:
623         // Initialization
624
625         l.mtspr     r0, r0, SPR_PCCR(6)
626         l.addi      r3, r0, (SPR_PCMR_CISM | SPR_PCMR_CIUM)
627         l.ori r3, r3, SPR_PCMR_BS
628         l.mtspr     r0, r3, SPR_PCMR(6)
629         // Start Counting //
630
631         // When the jump or branch is in EX stage and it's taken, the new
632         // instruction will be available after 2 cycles. In the first one,
633         // there is a valid instruction in the delay slot. In the second
634         // one, there aren't more valid instructions and a nop is inserted.
635         // So every time that a branch or jump is taken, a nop is inserted
636         // into the pipeline.
637
638         l.addi      r3, r0, 0xF // Number of loop iterations
639
640     _bsLOOP:
641         l.addi      r3, r3, -1
642         l.sfeqi     r3, 0x0
643         l.bnf _bsLOOP           // T x14
644         l.andi      r4, r3, 0x5 // DS
645
646         l.j         _bsS2       // T
647         l.addi      r4, r0, 0x1
648
649     _bsNOT:
650         l.j         _bsEND      // NT
651         l.addi      r7, r0, 0xDEAD
652
653     _bsS1:
654         l.or  r3, r4, r5
655         l.sfeqi     r3, 0x2
656         l.bf  _bsNOT            // NT
657         l.sfeqi     r3, 0x3
658         l.j         _bsEND      // T
659         l.and r6, r4, r5
660
661     _bsS2:
662         l.j         _bsS1       // T
663         l.addi      r5, r0, 0x2
664
665     _bsEND:
666         // Finish Counting //
667         l.mtspr     r0, r0, SPR_PCMR(6)
668
669         // Result = 15(its)-1 + 3 = 17 = 0x11
670
671         l.mfspr     r26, r0, SPR_PCCR(6) // Read counter to GPRs
672
673         l.jr  r9 //return
674         l.nop
675   //
676   ////////////////////////////////////////////////////////////////
677
```

```
678  ////////////////////////////////////////////////////////////////////
679  // Load Store Unit Stall Event test
680  //
681  _LSUS_test:
682        // Cycles when a store or load is performed
683        // Initialization
684
685        l.mtspr     r0, r0, SPR_PCCR(7)
686        l.addi      r3, r0, (SPR_PCMR_CISM | SPR_PCMR_CIUM)
687        l.ori r3, r3, SPR_PCMR_LSUS
688        l.mtspr     r0, r3, SPR_PCMR(7)
689        // Start Counting //
690
691        // By definition, it is considered that LSU stalls when store or load
692        // is in EX stage and unstalls after ack.
693        // That happens at the same time that Wishbone bus serves instructions
694        // and thus, it is stalled 4 cycles (without caches).
695        // Additional cycles are added depending on the memory area. Check signals
696        // with wave viewer if the result changes. These hidden cycles are
697        // correct but not easy to determine through code.
698
699        l.addi      r4, r0, 0x5000
700        l.addi      r5, r0, 0x5
701        l.addi      r6, r0, 0x6
702        l.addi      r7, r0, 0x7
703        l.addi      r8, r0, 0x8
704        l.addi      r8, r0, 0x8
705        l.addi      r8, r0, 0x8
706        l.addi      r3, r0, 0x8
707
708    _lsusLOOP:
709        l.sw  0x0(r4), r5
710        l.sw  0x4(r4), r6
711        l.sw  0x8(r4), r7
712        l.sw  0xC(r4), r8
713        l.lwz r8, 0x0(r4)
714        l.lwz r7, 0x4(r4)
715        l.lwz r6, 0x8(r4)
716        l.lwz r5, 0xC(r4)
717        l.addi      r4, r0, 0x40
718        l.addi      r5, r0, 0x5
719        l.addi      r6, r0, 0x6
720        l.addi      r7, r0, 0x7
721        l.addi      r3, r3, -1
722        l.sfeqi     r3, 0x0
723        l.bnf _lsusLOOP
724        l.addi      r8, r0, 0x8
725
726        // Finish Counting //
727        l.mtspr     r0, r0, SPR_PCMR(7)
728
729        // Result = (8MEMinstr * 4cycles * 8its.) = 256  = 0x100 (base)
730        //                                    +8          = 0x108
731        //                                       +8 +8   = 0x110
732        //                                                = 0x124
733
734        // These extra cycles are from wishbone bus to memory. Check wave viewer
735
736        l.mfspr     r27, r0, SPR_PCCR(7) // Read counter to GPRs
737
738        l.jr  r9 //return
739        l.nop
740  //
```

```
741    ////////////////////////////////////////////////////////////////////
742
743    ////////////////////////////////////////////////////////////////////
744    // Data TLB Miss Event test
745    //    Based on original or1200 mmu test
746    //
747    _DTLBM_test:
748          // Initialization
749          // 8Mb physical SDRAM - ITLB 1Way, Page Size = 8Kb, 64 TLB sets
750
751          l.addi      r3, r0, 0x2000    // 8192, Page Size
752          l.addi      r7, r0, 0x40      // 64, Number of DTLB sets
753          l.addi      r5, r0, 0xD       // 13, log2(Page Size)
754
755          l.sw  0x40(r0), r9      // Store link register
756
757          // Disable DMMU
758          l.mfspr r6, r0, SPR_SR
759          l.addi  r4, r0, -1
760          l.xori  r4, r4, SPR_SR_DME
761          l.and   r4, r6, r4
762          l.mtspr r0, r4, SPR_SR
763
764          // Invalidate all entries in DTLB
765          l.addi      r6, r0, 0x0
766      .invDTLB_Loop:
767          l.mtspr     r6, r0, SPR_DTLBMR_BASE(0)
768          l.mtspr     r6, r0, SPR_DTLBTR_BASE(0)
769          l.addi      r6, r6, 0x1
770          l.sfne      r6, r7
771          l.bf  .invDTLB_Loop
772          l.nop
773
774          // Set the botom MMU page (and thus TLB set) that test will begin at,
775          // (hopefully) avoiding pages with program text, data and stack,
776          // determining one page after the page where top of stack is on.
777                .global _stack
778          l.movhi     r1, hi(_stack)          // end data addr
779          l.ori r1, r1, lo(_stack)
780          l.srl r4, r1, r5  // Page Number of stack address
781
782                .global _stext
783          l.movhi     r2, hi(_stext)          // start text addr
784          l.ori r2, r2, lo(_stext)
785
786          // r4 contains first page number to use in this test: start_page_test
787          l.addi      r4, r4, 1   // Next page from stack's page number
788
789          // Set one to one translation for pages of program text, data and stack
790          // to make sure that they don't cause a miss.
791          l.addi      r6, r0, 0x0
792          l.addi      r8, r0, 0x0 // just for bits [31:13], DTLBTR and DTLBMR
793      _1t1Dtrans:
794          l.ori r10, r8, SPR_DTLBMR_V
795          l.mtspr     r6, r10, SPR_DTLBMR_BASE(0) // ea | SPR_DTLBMR_V
796          l.ori r10, r8, DTLB_PR_NOLIMIT
797          l.mtspr     r6, r10, SPR_DTLBTR_BASE(0)   // ta | DTLB_PR_NOLIMIT
798          l.addi      r6, r6, 0x1
799          l.sfeq      r6, r4
800          l.bnf _1t1Dtrans
801          l.addi      r8, r8, 0x2000
802
803          // SDRAM of 8MB, go far enough to put instructions there
```

```
804        l.movhi     r13, 0x0040 // Start from middle
805
806        l.addi      r14, r4, 0x3   // # of page to go. 3 more after stack's page
807        l.sll r14, r14, r5      // PageNum * PageSize
808
809        l.add r15, r13, r14     // TRANSLATE space = 0x00400000 + pageNum*PageSize
810
811        l.movhi     r16, 0x0011
812        l.ori r16, r16, 0x2233
813        l.sw  -4(r15), r16      // PageNum = start_page_test + 2; M
814
815        l.movhi     r16, 0x4455
816        l.ori r16, r16, 0x6677
817        l.sw  0(r15), r16       // PageNum = start_page_test + 3; M
818
819        l.addi      r15, r15, 0x2000 // Increment one page
820        l.movhi     r16, 0x8899
821        l.ori r16, r16, 0xAABB
822        l.sw  -4(r15), r16      // PageNum = start_page_test + 3; H
823
824        l.movhi     r16, 0xCCDD
825        l.ori r16, r16, 0xEEFF
826        l.sw  0(r15), r16       // PageNum = start_page_test + 4; M
827
828        l.addi      r15, r15, 0x2000 // Increment one page
829        l.movhi     r16, 0xBAAA
830        l.ori r16, r16, 0xAAAD
831        l.sw  0(r15),     r16         // PageNum = start_page_test + 5; M
832
833        // Space we'll access and expect the MMU to translate our requests
834        l.slli      r15, r3, 0x6      // 6 = log2(# of DTLB sets); sets*pageSize
835        l.add r15, r15, r14     // MATCH space = sets*pageSize + pageNum*PageSize
836
837        // Enable DMMU
838      l.mfspr r6, r0, SPR_SR
839      l.ori   r6, r6, SPR_SR_DME
840      l.mtspr r0, r6, SPR_SR
841      l.mtspr r0,r9,SPR_EPCR_BASE
842
843        // Enable counter
844        l.mtspr     r0, r0, SPR_PCCR(0)
845        l.addi      r3, r0, (SPR_PCMR_CISM | SPR_PCMR_CIUM)
846        l.ori r3, r3, SPR_PCMR_DTLBM
847        l.mtspr     r0, r3, SPR_PCMR(0)
848        // Start Counting //
849
850        // Check DTLB miss handler above, which will put EA-TA pair
851        l.lwz r17, -4(r15)      // Read last address of previous page
852                                // (MATCH space -4)
853        l.movhi     r16, 0x0011       // 1st MISS
854        l.ori r16, r16, 0x2233
855        l.sfne      r16, r17
856        l.bf  _dtlbmEND
857        l.nop
858
859        l.lwz r17, 0(r15)       // Read first address of the page (MATCH space)
860        l.movhi     r16, 0x4455       // 2nd MISS
861        l.ori r16, r16, 0x6677
862        l.sfne      r16, r17
863        l.bf  _dtlbmEND
864        l.nop
865
866        l.addi      r15, r15, 0x2000 // Increment one page
```

```
867
868         l.lwz r17, -4(r15)      // Read last addr of the page
869                                 // (MATCH space + pageSize -4)
870         l.movhi     r16, 0x8899       // No miss. Exception of 2nd miss will
871                                       // put EA-TA properly
872         l.ori r16, r16, 0xAABB
873         l.sfne      r16, r17
874         l.bf  _dtlbmEND
875         l.nop
876
877         l.lwz r17, 0(r15)       // Read first address of next page
878                                 // (MATCH space + pageSize)
879         l.movhi     r16, 0xCCDD       // 3nd MISS
880         l.ori r16, r16, 0xEEFF
881         l.sfne      r16, r17
882         l.bf  _dtlbmEND
883         l.nop
884
885         l.addi      r15, r15, 0x2000 // Increment one page
886
887         l.lwz r17, 0(r15)       // Read first address of 2nd next page
888                                 // (MATCH space + 2pageSize)
889         l.movhi     r16, 0xBAAA       // 4th MISS
890         l.ori r16, r16, 0xAAAD
891
892     _dtlbmEND:
893
894         // Finish Counting //
895         l.mtspr     r0, r0, SPR_PCMR(0)
896
897         // Disable DMMU
898         l.mfspr r6, r0, SPR_SR
899         l.addi  r4, r0, -1
900         l.xori  r4, r4, SPR_SR_DME
901         l.and   r4, r6, r4
902         l.mtspr r0, r4, SPR_SR
903
904         // Result = 4 misses = 0x4
905
906         l.mfspr     r28, r0, SPR_PCCR(0) // Read counter to GPRs
907
908         l.lwz r9,   0x40(r0)    // Restore link register
909         l.jr  r9 //return
910         l.nop
911   //
912   /////////////////////////////////////////////////////////////////
913
914   /////////////////////////////////////////////////////////////////
915   // Instruction TLB Miss Event test
916   //    Based on original or1200 mmu test
917   //
918   _ITLBM_test:
919         // Initialization
920         // 8Mb physical SDRAM - ITLB 1Way, Page Size = 8Kb, 64 TLB sets
921
922         l.addi      r3, r0, 0x2000    // 8192, Page Size
923         l.addi      r7, r0, 0x40      // 64, Number of ITLB sets
924         l.addi      r5, r0, 0xD       // 13, log2(Page Size)
925
926         l.sw  0x40(r0), r9      // Store link register
927
928         // Disable IMMU
929         l.mfspr r6, r0, SPR_SR
```

```
930         l.addi  r4, r0, -1
931         l.xori  r4, r4, SPR_SR_IME
932         l.and   r4, r6, r4
933         l.mtspr r0, r4, SPR_SR
934 
935         // Invalidate all entries in ITLB
936         l.addi      r6, r0, 0x0
937     .invITLB_Loop:
938         l.mtspr     r6, r0, SPR_ITLBMR_BASE(0)
939         l.mtspr     r6, r0, SPR_ITLBTR_BASE(0)
940         l.addi      r6, r6, 0x1
941         l.sfne      r6, r7
942         l.bf  .invITLB_Loop
943         l.nop
944 
945         // Set the botom MMU page (and thus TLB set) that test will begin at,
946         // (hopefully) avoiding pages with program text, data and stack,
947         // determining one page after the page where top of stack is on.
948               .global _stack
949         l.movhi     r1, hi(_stack)          // end data addr
950         l.ori r1, r1, lo(_stack)
951         l.srl r4, r1, r5  // Page Number of stack address
952 
953               .global _stext
954         l.movhi     r2, hi(_stext)          // start text addr
955         l.ori r2, r2, lo(_stext)
956 
957         // r4 contains first page number to use in this test: start_page_test
958         l.addi      r4, r4, 1   // Next page from stack's page number
959 
960         // Set one to one translation for pages of program text, data and stack
961         // to make sure that they don't cause a miss.
962         l.addi      r6, r0, 0x0
963         l.movhi     r8, hi(SPR_ITLBTR_PPN)
964         l.ori r8, r8, lo(SPR_ITLBTR_PPN)
965         l.and r8, r2, r8 // just for bits [31:13], ITLBTR and ITLBMR
966     _1t1Itrans:
967         l.ori r10, r8, SPR_ITLBMR_V
968         l.mtspr     r6, r10, SPR_ITLBMR_BASE(0) // ea | SPR_ITLBMR_V
969         l.ori r10, r8, ITLB_PR_NOLIMIT
970         l.mtspr     r6, r10, SPR_ITLBTR_BASE(0)   // ta | ITLB_PR_NOLIMIT
971         l.addi      r6, r6, 0x1
972         l.sfeq      r6, r4
973         l.bnf _1t1Itrans
974         l.addi      r8, r8, 0x2000
975 
976         // Write program which will just return to actual program (when they are
977         // called). Machine code for 'l.jr r9' and 'l.nop':
978         // Instr.   l.jr r9  => 0x44004800
979         l.movhi     r16, 0x4400
980         l.ori r16, r16, 0x4800
981 
982         // Instr.   l.nop    => 0x15000000
983         l.movhi     r17, 0x1500
984 
985         // SDRAM of 8MB, go far enough to put instructions there
986         l.movhi     r13, 0x0040 // Start from middle
987         l.ori r13, r13, 0x0
988 
989         l.addi      r14, r4, 0x3   // # of page to go. 3 more after stack's page
990         l.sll r14, r14, r5      // PageNum * PageSize
991 
992         l.add r15, r13, r14     // TRANSLATE space = 0x00400000 + pageNum*PageSize
```

```
993
994         l.sw  -8(r15), r16      // PageNum = start_page_test + 2; M
995         l.sw  -4(r15), r17
996         l.sw  0(r15), r16       // PageNum = start_page_test + 3; M
997         l.sw  4(r15),     r17
998         l.addi      r15, r15, 0x2000 // Increment one page
999         l.sw  -8(r15), r16      // PageNum = start_page_test + 3; H
1000        l.sw  -4(r15), r17
1001        l.sw  0(r15), r16       // PageNum = start_page_test + 4; M
1002        l.sw  4(r15),     r17
1003
1004        // Space we'll access and expect the MMU to translate our requests
1005        l.slli      r16, r3, 0x6      // 6 = log2(# of ITLB sets); sets*pageSize
1006        l.add r16, r16, r14     // MATCH space = sets*pageSize + pageNum*PageSize
1007
1008        // Enable IMMU
1009      l.mfspr r6, r0, SPR_SR
1010      l.ori   r6, r6, SPR_SR_IME
1011      l.mtspr r0, r6, SPR_SR
1012
1013        // Enable counter
1014        l.mtspr     r0, r0, SPR_PCCR(1)
1015        l.addi      r3, r0, (SPR_PCMR_CISM | SPR_PCMR_CIUM)
1016        l.ori r3, r3, SPR_PCMR_ITLBM
1017        l.mtspr     r0, r3, SPR_PCMR(1)
1018        // Start Counting //
1019
1020        // Check ITLB miss handler above, which will put EA-TA pair
1021
1022        l.addi      r17, r16, -8
1023        l.jalr      r17         // Jump on last address of previous page
1024                                // (MATCH space -8)
1025        l.nop             // 1st MISS
1026
1027        l.jalr      r16         // Jump on first address of the page (MATCH space)
1028        l.nop             // 2nd MISS
1029
1030        l.addi      r16, r16, 0x2000
1031        l.addi      r17, r16, -8
1032        l.jalr      r17         // Jump on last addr of the page
1033                                // (MATCH space + pageSize -8)
1034        l.nop          // No miss. Exception of 2nd miss will put EA-TA properly
1035
1036        l.jalr      r16         // Jump on 1st addr of next page
1037                                // (MATCH space + pageSize)
1038        l.nop             // 3rd MISS
1039
1040        // Finish Counting //
1041        l.mtspr     r0, r0, SPR_PCMR(1)
1042
1043        // Disable IMMU
1044        l.mfspr r6, r0, SPR_SR
1045        l.addi  r4, r0, -1
1046        l.xori  r4, r4, SPR_SR_IME
1047        l.and   r4, r6, r4
1048        l.mtspr r0, r4, SPR_SR
1049
1050        // Result = 3 misses = 0x3
1051
1052        l.mfspr     r29, r0, SPR_PCCR(1) // Read counter to GPRs
1053
1054        l.lwz r9,   0x40(r0)    // Restore link register
1055        l.jr  r9 //return
```

```
1056        l.nop
1057  //
1058  ///////////////////////////////////////////////////////////////////
1059
1060  ///////////////////////////////////////////////////////////////////
1061  // Memory accesses test
1062  //
1063  _SALA_test:
1064        // Initialization
1065
1066        // SPR_SR_SM: 0=UM, 1=SM
1067
1068        // Put CPU in Supervisor Mode
1069        l.mfspr     r2, r0, SPR_SR
1070        l.ori r2, r2, SPR_SR_SM
1071        l.mtspr     r0, r2, SPR_SR
1072
1073        // Set mask to turn CPU in User Mode
1074        l.addi  r1, r0, -1
1075        l.xori  r1, r1, SPR_SR_SM
1076
1077        // Put some values
1078        l.addi      r4, r0, 0x5000
1079        l.addi      r5, r0, 0x5
1080        l.addi      r6, r0, 0x6
1081        l.addi      r7, r0, 0x7
1082        l.addi      r8, r0, 0x8
1083
1084        // Enable counter
1085        l.mtspr     r0, r0, SPR_PCCR(2)
1086        l.addi      r3, r0, SPR_PCMR_CIUM   // Only count in User Mode
1087        l.ori r3, r3, (SPR_PCMR_LA | SPR_PCMR_SA) // Count 2 events
1088        l.mtspr     r0, r3, SPR_PCMR(2)
1089        // Start Counting //
1090
1091        //l.addi    r3, r0, 0x8
1092
1093        l.sw  0x0(r4), r5 // We are in SM, these accesses are not counted
1094        l.sw  0x0(r4), r5
1095        l.lwz r8, 0x0(r4)
1096
1097        // Put CPU in User Mode
1098        l.mfspr     r2, r0, SPR_SR
1099        l.and r2, r2, r1
1100        l.mtspr     r0, r2, SPR_SR
1101
1102        l.sw  0x4(r4), r6
1103        l.sw  0x8(r4), r7
1104        l.sw  0xC(r4), r8
1105        l.lwz r7, 0x4(r4)
1106        l.lwz r6, 0x8(r4)
1107        l.lwz r5, 0xC(r4)
1108
1109        // Trigger sys call exception to enable supervisor mode again (in user
1110        // mode SR cannot be read :D)
1111        l.sys 0
1112
1113        // Finish Counting //
1114        l.mtspr     r0, r0, SPR_PCMR(2)
1115
1116        // Result = 3 stores (UM) + 3 loads (UM) = 6 = 0x6
1117
1118        l.mfspr     r30, r0, SPR_PCCR(2) // Read counter to GPRs
```

```
1119
1120        l.jr  r9 //return
1121        l.nop
1122  //
1123  ////////////////////////////////////////////////////////////////////
1124
1125  ////////////////////////////////////////////////////////////////////
1126  // Combined test
1127  //
1128  _combined_test:
1129        // Initialization
1130
1131        // Clear counters used in this test
1132        l.mtspr     r0, r0, SPR_PCCR(3)
1133        l.mtspr     r0, r0, SPR_PCCR(4)
1134        l.mtspr     r0, r0, SPR_PCCR(5)
1135        l.mtspr     r0, r0, SPR_PCCR(6)
1136        l.mtspr     r0, r0, SPR_PCCR(7)
1137
1138        // IC of 4KB, 256 sets, 16 bytes/set
1139        // Instructions of 4 bytes.
1140        // 4 instr/set --> 1 miss every 4 instructions
1141        l.addi      r14, r0, 0x10     // 16, Cache block size
1142        l.addi  r5, r0, 0x8           // 8, log2(# of cache sets)
1143        l.addi   r7, r0, 0x100  // 256, Number of cache sets
1144
1145        // Disable IC
1146        l.mfspr     r6, r0, SPR_SR
1147        l.addi      r3, r0, -1
1148        l.xori      r3, r3, SPR_SR_ICE
1149        l.and r3, r6, r3
1150        l.mtspr     r0, r3, SPR_SR
1151
1152        // Invalidate IC
1153        l.addi      r6, r0, 0x0
1154        l.sll   r5, r14, r5
1155    .invLoop:
1156        l.mtspr     r0, r6, SPR_ICBIR
1157        l.sfne      r6, r5
1158        l.bf  .invLoop
1159        l.add r6, r6, r14
1160
1161        // Enable IC
1162        l.mfspr r6, r0, SPR_SR
1163        l.ori   r6, r6, SPR_SR_ICE
1164        l.mtspr r0, r6, SPR_SR
1165        l.nop
1166        l.nop
1167        l.nop
1168        l.nop
1169        l.nop
1170        l.nop
1171        l.nop
1172        l.nop
1173
1174        // Enable counters //
1175
1176        l.addi      r1, r0, (SPR_PCMR_CISM | SPR_PCMR_CIUM) //Count in SM & in UM
1177
1178        // Count Instruction fetches event in PCCR3
1179        l.ori r3, r1, SPR_PCMR_IF
1180        l.mtspr     r0, r3, SPR_PCMR(3)
1181
```

```
1182        //Count Instruction Cache misses event in PCCR4
1183        l.ori r3, r1, SPR_PCMR_ICM
1184        l.mtspr     r0, r3, SPR_PCMR(4)
1185
1186        //Count Load Access event in PCCR5
1187        l.ori r3, r1, SPR_PCMR_LA
1188        l.mtspr     r0, r3, SPR_PCMR(5)
1189
1190        //Count Branch Stalls event in PCCR6
1191        l.ori r3, r1, SPR_PCMR_BS
1192        l.mtspr     r0, r3, SPR_PCMR(6)
1193
1194        //Count Store Access event in PCCR7
1195        l.ori r3, r1, SPR_PCMR_SA
1196        l.mtspr     r0, r3, SPR_PCMR(7)
1197
1198        // All counters for this test are enabled yet //
1199
1200        // Some instructions
1201        l.addi      r10, r0, 0xA
1202        l.addi      r11, r0, 0xB
1203        l.addi      r12, r0, 0xC
1204        l.addi      r13, r0, 0xD
1205        l.addi      r14, r0, 0xE
1206        l.addi      r15, r0, 0xF
1207        l.addi      r16, r0, 0x10
1208
1209        l.j   _combNext
1210        l.nop
1211
1212  _comb_ldst:
1213        l.sw  0(r4), r2
1214        l.lbz r17, 0(r4)
1215        l.j   _combEND
1216        l.addi      r5, r0, 0x5
1217
1218  _combNext:
1219        l.add r6, r0, r11
1220        l.add r7, r0, r10
1221        l.add r8, r0, r12
1222        l.j   _comb_ldst
1223        l.addi      r4, r0, 0x4
1224
1225  _combEND:
1226        // Finish counting //
1227        l.mtspr     r0, r0, SPR_PCMR(3)
1228        l.mtspr     r0, r0, SPR_PCMR(4)
1229        l.mtspr     r0, r0, SPR_PCMR(5)
1230        l.mtspr     r0, r0, SPR_PCMR(6)
1231        l.mtspr     r0, r0, SPR_PCMR(7)
1232
1233  // Results
1234  //     Event            Result
1235  //    IF   -  PCCR3  - 27 instr (from l.ori ICM to l.mtspr IF) = 0x1b
1236  //    ICM  -  PCCR4  - 26 instr/4 (from l.ori LA to l.mtspr ICM) = 6 or 7 misses
1237  //    LA   -  PCCR5  - 1 load access                        => Result: 0x1
1238  //    BS   -  PCCR6  - 3 jumps or branchs                   => Result: 0x3
1239  //    SA   -  PCCR7  - 1 store access                       => Result: 0x1
1240
1241        l.jr  r9    // return
1242        l.nop
1243  //
1244  //////////////////////////////////////////////////////////
```

D.2 Descripció del joc de proves

Com que hi ha més esdeveniments que comptadors, el *main* s'ha desglossat en 2 tests per poder comprovar tots aquests tests. En concret, el primer test executa els test IF, LA , SA, DCM, ICM, IFS, BS i LSUS individualment. El segon test comprova individualment els esdeveniments DTLBM i ITLBM, comprova com un comptador pot estar comptant més d'un esdeveniment i juntament comprova si el comptatge es realitza en mode usuari o sistema. Finalment, s'executa un test conjunt que habilita més d'un comptador a la vegada i fa alguna comprovació més amb un codi nou.

Esdeveniment IF (línies 236 – 272)

S'inicialitza el comptador per comptar en mode usuari i sistema, i l'esdeveniment IF. A partir d'aquí s'executa un conjunt d'instruccions, 2 instruccions soles, i un bucle de 7 instruccions que s'itera 12210 vegades. El resultat és la suma d'aquests valors comptant-hi també una instrucció més, la que desactiva el comptador, que al tractar-se de *fetchs* aquesta instrucció ja ha estat comptada.

Esdeveniment LA (274 – 341)

S'inicialitza el comptador, i hi ha 8 loads dins del bucle que s'itera 10 cops, per tant, el resultat són de 80 accessos deguts a loads. També es realitzen accessos d'escriptura a memòria, però es pot apreciar com no estan comptabilitzats.

Esdeveniment SA (343 – 404)

Similarment a l'esdeveniment anterior, es proveeix un codi amb 7 stores dins d'un bucle que s'itera 20 vegades. El resultat és de 140 accessos store. Igualment, comprovar que els accessos de lectura a memòria no incrementen aquest comptador.

Esdeveniment DCM (406 – 471)

Per fer aquest test, primer cal activar la cache de dades, doncs no està activada. Per això s'invaliden totes les línies i després s'activa la cache. Acte seguit s'activa el comptador i tal com s'ha programat el test, fa un store, que al tractar-se del primer accés falla. El que passa és que la cache està configurada com a write-through i per tant, quan l'store ha fallat, ha accedit a memòria principal però no ha carregat la línia a cache. Per això, al fer un load de la mateixa adreça, torna a fer un miss, però ara ja sí que es porta la línia, i per tant, els altres accessos a la mateixa línia encerten.

En una línia de cache hi caben 4 dades de 32 bits, per tant, dins el bucle es fan accessos als dos primers bytes, i l'adreça s'incrementa de 2 bytes en 2 bytes a cada iteració. Això fa que només 1 de cada 2 iteracions falli a cache 2 cops (el primer store i el primer load). Per això, el resultat és la meitat del nombre d'iteracions per aquests dos missos: 220 missos.

Esdeveniment ICM (473 – 552)

Igualment com a l'esdeveniment anterior, primer s'ha d'invalidar les línies de la cache d'instruccions i activar-la. Després s'activa el comptador i es comprova un codi. El codi està format per grups de 4 instruccions, ja que al disposar de línies de 16 bytes, a cada línia hi cap 4 instruccions i, per tant, cada 4 instruccions una fallarà. Un d'aquests grups de 4 s'ha encapsulat dins d'un bucle per demostrar que com que la línia ja conté instruccions vàlides, no provoquen missos.

Així doncs, el test consta de 4 grup de 4 instruccions (l'últim grup, que aparentment només té 3 instruccions, és de 4 ja que també es compta el *fetch* de la instrucció que atura el comptador). El resultat doncs, és de 4 missos.

Esdeveniment IFS (554 – 617)

S'activa el comptador i es posen instruccions. Els cicles d'aturada depenen de la memòria. Cada 8 instruccions, s'atura 8 cicles per portar les instruccions de memòria al controlador de la memòria. A partir de llavors, cada 4 cicles el controlador va enviant una instrucció que arriba al pipeline. Això significa que cíclicament (sense cache activada) hi ha 8 cicles + 3*8 cicles d'aturada per cada 8 instruccions.

Si el codi presenta salts que salten, el processador insereix una NOP. Aquesta NOP és considera com una instrucció, perquè el PC sempre té valors, i per tant, durant la NOP també hi ha els 3 cicles de pausa on el controlador està portant una instrucció (no és que sigui una de les millor implementacions, però el processador funciona així).

A més a més, la memòria principal utilitzada és una SDRAM. Això significa que existeixen cicles de refresc de la memòria que aturen l'execució i són cicles que no es poden determinar tan sols amb el joc de proves, cal un anàlisi de les senyals per comprovar que el funcionament és correcte.

Esdeveniment BS (619 – 676)

Cada cop que un salt salta (és *taken*), s'executa una instrucció en DS, però també s'insereix una NOP. Per tant, cada cop que el salt sigui *taken* es comptarà un cicle de BS. En el codi es proven diferents salts i un que no s'executa. Per tant, el resultat és el nombre de salts *taken*: 17 (un d'ells està dins d'un bucle de 15 iteracions, que salta 14 vegades).

Esdeveniment LSUS (678 – 741)

Com ja s'ha parlat a la descripció de l'esdeveniment, els cicles en què la LSU està aturada són comptats des que un accés a memòria es troba en etapa EX fins que arriba l'ACK de memòria. Aquests cicles indiquen els cicles durant el quals la unitat LSU està ocupada. Mentre passa aquest esdeveniment, el bus *Wishbone* està servint instruccions de dades.

Així doncs, el nombre de cicles pot variar segons l'estat actual de memòria, segons els accessos per instruccions que s'estiguin fent i segons l'àrbitre del bus. Una comprovació de l'estat de les senyals podrà oferir una informació més precisa que analitzant el joc de proves, on tots aquests cicles ocults no es veuen.

Esdeveniment ITLBM (914 – 1058)

Per poder provar fallades de TLB, és necessària certa configuració inicial. Les pàgines de la TLB són de 8Kb i n'hi ha 64. El primer que es fa és desactivar la IMMU (només per si estigués activada) i invalidar totes les entrades. A partir d'aquí, hi ha una porció de codi per determinar a quina pàgina acaba hipotèticament la secció 'text', 'data' i 'stack' del programa. Per seguretat, al valor obtingut s'incrementa en 1 i és la pàgina que s'utilitzarà com a base per començar el test. Així doncs, totes les pàgines, des de l'actual fins a la 0 (corresponents a la part del programa) són introduïdes a la ITLB perquè aquestes pàgines no ocasionin fallades. A partir d'aquest moment, en àrees allunyades de memòria (a partir de la meitat de la memòria total disponible), s'insereix el codi màquina de dues instruccions: un salt al registre R9 (que és el registre de linkatge) i una NOP (com a DS).

Això permetrà, des de codi, cridar a aquestes posicions allunyades amb una instrucció "`l.jal`" perquè es guardi l'adreça de la següent instrucció al registre de linkatge R9, perquè el programa retorni al punt d'execució on s'estava fins ara.

Així doncs, ja només queda activar la IMMU, activar el comptador i saltar a aquestes pàgines. El primer salt generarà una excepció ja que no es troba la pàgina a la ITLB. Per això, també cal implementar la rutina de tractament de l'excepció causada per missos de ITLB, que el que fa és introduir l'entrada de l'adreça efectiva i l'adreça traduïda a la TLB perquè la trobi, i retorna des de l'excepció.

Es torna a saltar i torna a fallar, però el tercer salt correspon a la mateixa pàgina i es pot comprovar com no falla ja que al tractar l'excepció al salt anterior, ja ha introduït la pàgina. Finalment es torna a saltar a una pàgina diferent comprovant que falla de nou.

Per tant, amb tot això, s'obtenen 3 missos de ITLB.

Esdeveniment DTLBM (743 – 912)

Els passos a seguir per a les fallades de la TLB de dades és molt semblant al de l'esdeveniment anterior. Aquest codi es diferencia en que en comptes de posar instruccions en posicions allunyades de memòria, s'hi posen valors (amb stores) que després seran llegits amb loads.

S'accedeix a 4 pàgines diferents i un dels accessos accedeix a la mateixa pàgina. Per tant, s'obtenen 4 missos de DTLB. Cal destacar que també s'ha programat la rutina d'excepció de missos de DTLB.

Joc de prova LA+SA, amb CIUM i CISM (1060 – 1123)

El joc de proves en si mateix no presenta cap problema. La gràcia està en el fet que no es compten quan s'està en mode sistema, ja que s'ha programat el comptador perquè només compti en mode usuari. Per fer això, només cal desactivar el bit SM del registre SR, però un cop desactivat i es passa a mode usuari, ja no és accessible de nou per poder tornar a mode sistema, ni tampoc es pot aturar el comptador, ja que són registres especials que no són accessibles des de mode usuari.

Per poder tornar a mode sistema, s'ha implementat la rutina d'excepció de la crida a sistema, que posa el processador en mode sistema. Des de codi es crida amb la instrucció "`l.sys`".

Test amb varis comptadors (1125 – 1244)

Finalment, l'últim test no presenta res de nou, però és un test per comprovar el funcionament quan hi ha més d'un comptador activat alhora. Els resultats són fàcilment deduïbles des del mateix codi.

ANNEX E GUIA D'ERRORS

Tota l'estació de treball ha estat provada sota un sistema Linux: Ubuntu 12.04 de 32 bits.

Durant la realització del projecte s'han detectat errors i a continuació es detallen per si mai algú es troba en una situació semblant.

E.1 Mentre es programa la placa

Al connectar el cable entre la placa i l'ordinador i voler programar-hi el processador, pot aparèixer aquest error:
"*Error (213019): Can't scan JTAG chain. Error code 89.*"

Això és degut a que no s'ha iniciat correctament el dimoni Altera JTAG. El pas a seguir és reiniciar el dimoni des de la ruta on hi ha el software d'altera:

- `killall jtagd`
- `sudo /<ruta_altera>/quartus/bin/jtagd`

E.2 Simulant amb ModelSIM

Limitació versió gratuïta

En la versió gratuïta utilitzada de ModelSIM, no es permet tenir el codi en diferents llenguatges HDL. S'ha traduït el fitxer "altera_virtual_jtag.vhl" a Verilog.

Utilització de senyals abans de la seva declaració

Els fitxers "or1200_du.v", "or1200_genpc.v" i "orpsoc_top.v" tenen senyals accedides abans de la seva declaració. Tant la compilació com el simulador Icarus Verilog no presenten cap problema, però el ModelSIM es queixa i s'ha hagut de recol·locar les declaracions de les senyals.

Diferència entre instanciació de mòduls

El *dut* ("orpsoc_top.v") del fitxer de *testbench* que l'instancia amb el propi fitxer, tenen diferents mòduls activats i diferent nombre de senyals. S'ha hagut d'encaixar perquè coincideixin.

No apareixen fitxers de resultats

Per defecte, l'ORPSoC no incorpora el directori 'out' dins del directori 'sim'. Sense aquest directori no es creen els fitxers de sortida després de la simulació. Per solucionar-ho s'ha de crear aquest directori.

Errors de la simulació

 # Tap Controller State machine output error
 # Time: 0.0 ns Instance: orpsoc_testbench.dut.jtag_tap0.sld_virtual_jtag_component.jtag .output_logic

Encara que explícitament indiqui que es tracta d'un error, és un avís que apareix a l'executar la simulació amb ModelSIM i no afecta a l'execució. Es pot menysprear.

"Error: cannot find "/<modelsim_path>/bin/../linux_rh60/vsim"."

Això succeeix perquè s'està comparant el 3.0 a `/<modelsim_path>/vco`:

```
case $utype in
  2.4.[7-9]*    vco="linux" ;;
  2.4.[1-9][0-9]*)    vco="linux" ;;
  2.[5-9]*)      vco="linux" ;;
  2.[1-9][0-9]*) vco="linux" ;;
  *)       vco="linux_rh60" ;;
esac
```

Per tant, cal afegir la regla "`3.[0-9]*) vco="linux" ;;`".

*"Error: cannot find /<modelsim_path>/bin/../linux/vopt make: *** [modelsim] Error 1 "*

Això passa perquè no és capaç de trobar "vopt" per optimitzar. Per solucionar-ho, en comptes de cridar a l'executable es pot passar com a argument al simulador. Per fer això, al Makefile ja està preparat, però cal definir l'atribut:

- MGC_NO_VOPT=1 a "`/boards/altera/de1/sim/bin/Makefile`"

La simulació s'engega però no sembla que simuli

Amb la versió gratuïta del ModelSIM ja avisa que el rendiment serà afectat negativament per a aquest projecte. Això significa que per una execució de 30 segons, pot ser que trigui 5 minuts sense apreciar canvis.

No obstant, segons la configuració dels Makefiles, la regla per defecte no carrega el contingut del test a la memòria, i per tant no executa res. Cal definir "PRELOAD_RAM=1" juntament amb la crida de la regla de simulació: `make rtl-test TEST=<test> VCD=1 PRELOAD_RAM=1`.