UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica

Integració d'una unitat de coma flota en procesador RISC-V

Treball Fi de Grau

Grau en Enginyeria Informàtica

AUTOR/A: Salvá Grimalt, Xavier

Tutor/a: Flich Cardo, José

CURS ACADÈMIC: 2022/2023

Resum

El joc d'instruccions RISC-V és un joc d'instruccions lliure que permet que se'l faja servir sense haver de pagar regalies. Això el fa especialment atractiu tant per l'acadèmia com per la indústria i és per això que està vivint un creixement ràpid els últims anys.

En aquestes línies, a aquest document es parlarà sobre com s'ha fet per afegir suport per les extensions F i D de RISC-V a un processador que implementa RV64IMA. En concret, es farà una exploració tant a nivell arquitectònic revisant per quins components hauran de passar les noves instruccions com a nivell microarquitectònic parant especial atenció a quins canvis s'han hagut de realitzar per tal d'obtenir les funcionalitats requerides per l'estàndard. En aquests sentits es comentaran les decisions de disseny preses i les tècniques d'optimització incloses per tal d'incrementar l'IPC que han permés pujar-lo un +90%.

Finalment, es farà una petita anàlisi de les tècniques de verificació que s'han fet servir. També es comentarà com s'han obtingut els resultats de rendiment.

Paraules clau: processador, RISC-V, coma flotant, CPU, microaquitectura

Resumen

El conjunto de instrucciones RISC-V es un juego de instrucciones de uso libre con el que se permite el uso sin la necesidad de pagar regalías. Eso lo hace especialmente atractivo para la academia, pero tambien para la indústria. Es por eso que durante los últimos años ha sufrido un incremento en su uso acelerado.

En estas líneas, este documento tratará sobre como se ha añadido soporte para las extensiones F y D de RISC-V sobre un procesador que implementa RV64IMA. En concreto, se llevará a cabo una exploración a nivel arquitectónico revisando por qué componentes pasarán las instrucciones para las que se pretende añadir soporte, como a nivel microarquitectónico poniendo especial atención a los cambios que se han tenido que incluir para obtener las funcionalidades que el estándar requiere. En este sentido se comentarán las decisiones de diseño tomadas, pero también las técnicas de optimización que se han aplicado que han permitido un incremento del IPC de hasta un +90 %.

Finalmente, se analizarán brevemente las técnicas de verificacion que se han usado. Tambien se comentará como se han obtenido los resultados de rendimiento.

Palabras clave: procesador, coma flotante

Abstract

RISC-V is a free instruction set that allows its use without the need of any royalties payment. This makes it specially attractive to be used in academia, as well as in industry. That's why it has seen a explosion in its use in the last few years.

In this sense this document will talk about how we added support for the F & D RISC-V extensions on top of a RV64IMA processor. About this there will be an exploration at an architectonic level reviewing the components instruccions will go through to achieve the needed support, as well as a microarchitectonic review looking at the changes needed to achieve the features the standard specification requires. On this topic there will be an extensive commentary about the design decisions taken as well as the optimization techniques added to allow an IPC increment reaching up to +90%.

Finally, there will be a brief analysis on the verification techniques that have been used. There will be as well a short commentary explaining how the performance results have been obtained.

Key words: processor, floating point, microarchitecture

Índex

Índex de figures

Índex de taules

Agraïments

En primer lloc agrair al *Barcelona Supercomputing Center — Centro Nacional de Supercomputacion* (BSC/CNS) per brindar-me la oportunitat de col·laborar amb el projecte DRAC, dins del marc d'unes pràctiques remunerades. Especialment, agrair-li a l'equip del grup de *Computer Architecture for Parallel Paradigms* per ajudar-me en la formació i el desenvolupament necessaris per aquest treball, però també per tot el que hem aprés junts

També els voldria agrair en concret el seu suport a César Alejandro Hernández Calderón, pel seu suport durant el desenvolupament del projecte i la redacció d'aquest document. Sobre aquest últim aspecte també s'ha de reconèixer la tasca que ha realitzat José Flich Cardo.

CAPÍTOL 1
Introducció

En aquest capítol s'introduirà la idea general que es desenvoluparà, la motivació, els factors que l'han justificat, i els objectius que s'esperen assolir en concloure el projecte.

1.1 Conceptes Previs

1.1.1. Canvis Arquitectònics

Desde l'aparició dels primers processadors de propòsit general, ha existit una necessitat d'incrementar la seua capacitat de càlcul. Això s'ha aconseguit millorant les tècniques per les quals s'implementen els circuits digitals, i desde l'aparició dels circuits integrats, incrementant el grau de miniaturització dels circuits, però també amb canvis en la seua estructura i funcionament lògic interns.

El processador més bàsic és el que anomenem el *processador monocicle*, que executa una instrucció per cicle i duu a terme totes les operacions necessàries en el mateix cicle, és a dir que no manté diverses instruccions en vol. El problema d'aquest sistema és que presenta un camí crític molt llarg i força a mantenir una freqüència d'operació molt baixa.

La solució típica a aquest problema és *segmentar* el processador, això és, ubicar les diverses unitats funcionals necessàries per l'execució de programes en fases diverses separades per registres que permeten fraccionar el camí crític i multiplicar la freqüència d'operació del sistema. Tot i això, aquesta idea no és perfecta, ja que quan es detecta una dependència de dades tot el sistema s'ha d'aturar fins que quede resolta. Això causa que en determinats programes les instruccions que s'executen per cicle (IPC) siga menor que el que es podria arribar a assolir en una execució ideal.

Per resoldre aquesta problemàtica, es va introduir l'execució fora d'ordre (*Out-of-Order Execution*, OoO), que permet executar instruccions fora d'ordre sempre que es respecten les dependències. D'aquesta manera es pot fer un millor aprofitament del *paral·lelisme a nivell d'instrucció* (ILP), però això implica incrementar la complexitat de la lògica de control introduint estructures auxiliars que poden incrementar significativament l'àrea de silici ocupada i el consum energètic. De totes maneres, amb codis amb moltes dependències (de dades o de control) aquesta tècnica tampoc és perfecta, és per això que els compil·ladors i el programador també tenen un paper important pel que fa al rendiment del codi a executar, però son qüestions que més bé s'han de resoldre cas per cas del codi. [6]

Afegint a les tècniques anteriorment esmentades, l'execució es pot accelerar introduint el concepte de *processadors superescalars* que permeten l'execució de més d'una instrucció al mateix cicle. De tal manera que un processador superescalar de quatre vies pot

completar l'execució de fins a 4 instruccions simultàniament. En aquest cas, el sistema fa ús del paral·lelisme a nivell de dades (DLP), tot i que en programes amb moltes instruccions de control de flux el rendiment que oferix aquesta tècnica queda limitat. Tot i això, és una tècnica molt potent per aplicacions en les que trobem intruccions amb un grau alt de paral·lelisme. [6]

Finalment, una tècnica que ens permet reduir més el temps d'execució és la de introduir en un mateix sistema diversos processadors, a això se li anomena un *Multiprocessador*. Realment a nivell intern del processador els canvis que implica un sistema multiprocessador no són massa grans, i el major canvi ve en l'establiment de protocols de coherència i consistència amb memòria. Això permet executar diverses tasques simultàniament (*Thread-level parallelism*, TLP), o resoldre'n una de gran més ràpid fent ús de diversos fils d'execució. Aquesta idea és especialment útil per grans aplicacions amb graus molt alts de paral·lelisme, i és la idea sobre la que es fonamenten els supercomputadors del món modern, que interconnecten grans quantitats de processadors (i altres elements de càlcul) amb la fi d'assolir volums de càlcul màxims. Fer ús d'aquest tipus de paral·lelisme necessita de programadors altament especialitzats que siguen conscients de com implementar codi per execució que fa ús del TLP.

Als últims anys també s'ha observat una tendència d'aparició d'acceleradors de càlcul específics per a aplicacions concretes. Els primers, més notables i més pròxims a l'usuari habitual són les targetes gràfiques (*Graphical Processing Unit*, GPU), que ofereixen la possibilitat de fer càlculs amb una lògica senzilla però amb molt de paral·lelisme. També és interessant esmentar les TPUs, pensades per l'entrenament de xarxes neuronals, que multipliquen matrius per hardware, en lloc de fer-ho per software com s'ha fet convencionalment. [6]

1.1.2. Ferramentes i Programari Lliure

L'increment en la complexitat de l'arquitectura dels processadors ha anat escalant a mesura que passaven els anys. Típicament aquesta idea està enllaçada a la Llei de Moore, per la que es duplica la densitat de transistors per unitat d'àrea cada dos anys. Això no és pas cap llei formal com ho puguen ser les lleis físiques, sinó més bé és el ritme al que s'aspira a millorar implementacions en el pas dels anys, és a dir que no és una qüestió que passe de manera natural i necessita del desenvolupament de ferramentes auxiliars que ens permeten fer un desenvolupament efectiu amb un nombre exponencialment major de transistors.

Inicialment els circuits lògics es dissenyaven a mà fent ús de paper i llapis, això no escala particularment bé quan parlem d'integrar milers de milions de transistors i és per això que es desenvoluparen ferramentes de *Disseny Automàtic d'Electrònica* (EDA), per tal de no sols oferir entorns en els que descriure circuits integrats, sinó també poder-los simular per garantir un correcte funcionament dels circuits integrats que s'imprimiran en el silici.

Un dels primers softwares amb aquest objectiu va ser *Simulation Program with Integrated Circuits Emphasis* (SPICE, 1973), un programari inicialment de domini públic (i posteriorment lliure) que permet simular circuits integrats a nivell de transistor (Physical level). D'aquest programari n'han sorgit d'altres que iteren sobre la seua funcionalitat millorant alguns aspectes i enfocant-los a aplicacions concretes. Val a dir que les comprovacions a nivell físic s'han convertit en un dels passos estàndard a la indústria abans d'enviar el disseny per ser fabricat. [7]

El problema amb les simulacions a nivell físic és que es preocupen de molts aspectes que en fases prèvies del desenvolupament no són conegudes (com puga ser la ubicació

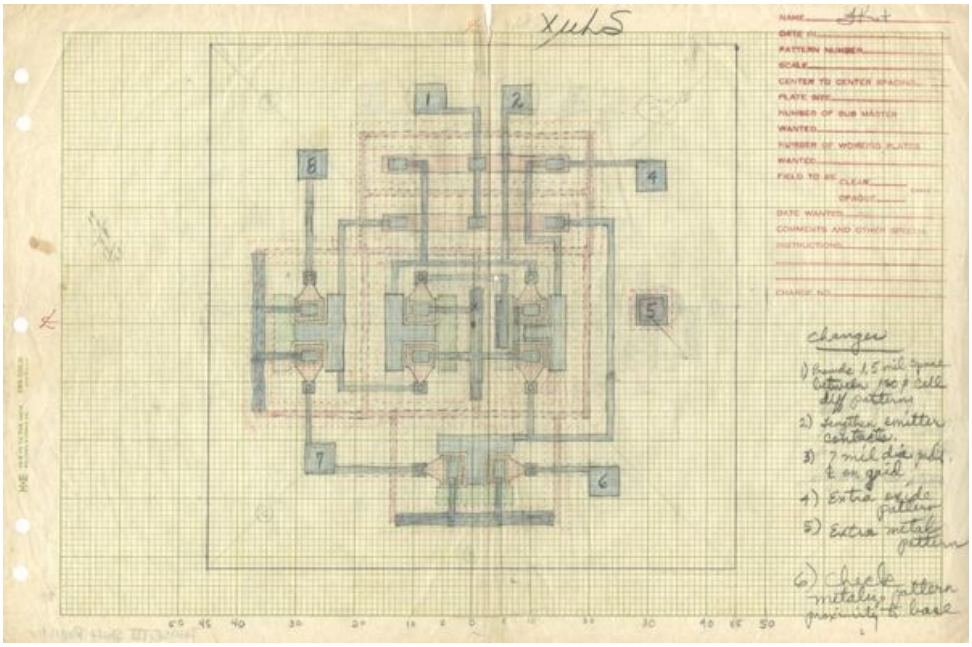


Figura 1.1: Exemple de la implementació física d'un circuit feta a mà. Fairchild, 1960

dels components lògics dins del global del circuit). A més, aquestes simulacions, en circuits més grans solen ser computacionalment pesants, pel que no són ideals per cicle de desenvolupament i prototipat ràpid que permet anar generant components amb un alt grau de correcció lògica.

És per això que es construïren els *Llenguatges de descripció de Hardware* (HDL), que permeten descriure circuits lògics fent ús d'una interfície similar a la d'un llenguatge de programació, i permeten un flux de treball semblant al que és possible amb llenguatges de programació típics, de codificació i execució amb tests. Els dos HDL més emprats són *VHSIC Hardware Description Language* (VHDL) i Verilog, tots dos apareguts durant la dècada dels 80, han fet possible el disseny de circuits integrats amb gran quantitat de cel·les de memòria i circuits combinacionals.

Però tot això genera un problema i és que pel fet de codificar un circuit no necessàriament estarà bé (i no sol estar-ho), així que en hem de plantejar com verificar que el circuit dissenyat és lògicament correcte, és a dir, que compleix la funció que inicialment s'havia planejat, amb tests lògics o funcionals. Per això existeixen els simuladors lògics. N'hi ha molts i de molt diversos, i permeten compil·lar la descripció en un HDL i simular-la. Hui en dia n'existeixen molts, però potser els que presenten un ús més comú són l'XSIM de Xilinx i el ModelSim de Siemens, tots dos amb llicències privatives, tot i que ofereixen versions específiques per estudiants i entusiastes per tal que es puguen introduir al món del disseny lògic, tot amb característiques limitades.

Existeixen alternatives lliures, com puga ser Verilator, un programari que compil·la descripcions HDL a C++. Aquest codi es pot posteriorment modificar i compil·lar per tenir un control sobre què ocorre en cada punt del circuit per codi.

Aquest control és important per un aspecte central del desenvolupament de hardware, la verificació. Verificar tots els possibles estats d'un circuit és impossible pel creixement exponencial que presenta en relació al nombre de cel·les de memòria. Per aquesta raó, típicament es comproven, en primer lloc, tests d'estats típics. I una vegada es confirma el comportament adequat en circumstàncies normals, s'apliquen tests generats aleatòriament, ajustats a uns paràmetres segons les necessitats que s'han de verificar. Per exemple, per comprovar el comportament adequat d'un processador, en primer lloc es

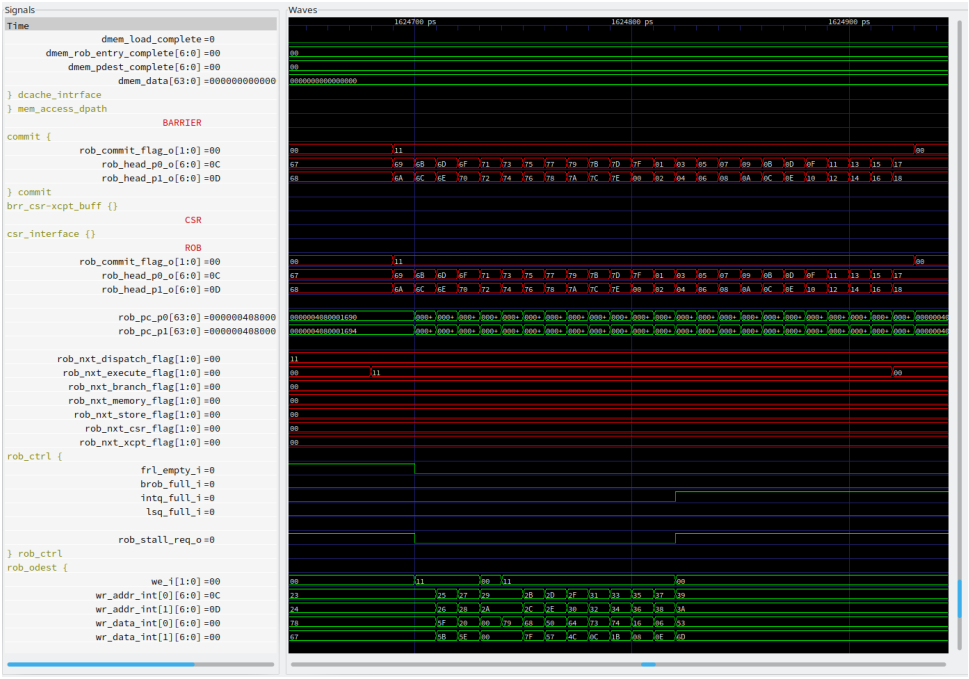


Figura 1.2: Exemple d'una traça d'ones

verifica el comportament normal en aplicacions típiques i tests estandarditzats. Tot seguit, es generen seqüències d'instruccions aleatòries que de per sí no tenen massa sentit, però que permeten trobar fallades més estranyes.

Moltes vegades, però, les comprovacions per codi son insuficients o complexes d'implementar, pel que els simuladors també ofereixen la possibilitat d'observar com canvien els senyals en cada punt del circuit d'una simulació amb l'avenç dels cicles. Se les anomena traces d'ones i són especialment útils a l'hora de debuggar determinats components que han quedat mal codificats i estan mostrant un comportament inadequat, i determinar exactament quin és el codi concret que causa problemes.

També és important esmentar una problemàtica recurrent al camp: moltes vegades en el desenvolupament de programari es prenen components desenvolupats per altres desenvolupadors per resoldre problemes concrets. Al desenvolupament de hardware això no és una idea tan comuna, ja que existeix una propensió important a blocar la compartició d'idees desenvolupades per un grup amb un altre que se'n podria beneficiar darrere de restriccions de clàusules de confidencialitat, regalies o altres mecanismes d'imposició de la propietat intel·lectual. Això fa que en molts projectes s'hagen de redissenyar components que ja existeixen, o haver de sol·licitar al propietari el seu ús. Totes dues alternatives poden arribar a ser força costoses (en temps i diners) i desincentiven l'aparició de nous competidors, especialment els més petits, al sector.

Afortunadament, als últims anys, iniciatives com RISC-V o l'Open Hardware Initiative estan contribuint a construir un marc de desenvolupament més lliure i que permeta l'entrada al mercat a nous competidors oferint nous productes lliures o amb llicències d'ús menys restrictives i obertes.

1.1.3. Sobirania Tecnològica

Una altra qüestió que està fent que la indústria del desenvolupament hardware siga més oberta és la demanda per sobirania tecnològica de diversos països arreu del món com puguen ser els EUA, la Xina, o el bloc de la UE.

Desde els anys 80, les indústries de països desenvolupats han patit un procés de deslocalització del seu país natal a un en vies de desenvolupament. Això en un principi va estar motivat pels sous baixos, que permetien abaratir la producció de manera significativa. En el cas de la indústria del semiconductor això ha acabat en la consolidació de grans empreses productores que imprimeixen els microxips dissenyats en silici ultrapur en fàbriques-laboratori (*Fabs*), centres amb tecnologia puntera incapaç de ser replicada a cap altre lloc al món. Aquest va estar un sistema que va funcionar sense problemes (però no sense crítics) fins al 2020.

Aquell any va venir la pandèmia de la COVID-19, que va fer que una gran part de la població havera de deixar de treballar per poder atendre les restriccions sanitàries que es van imposar. Això va provocar una aturada de cadenes de subministrament arreu de l'economia. Per exemple, a la indústria automobilística es van cancel·lar les peticions de xips, ja que, mentre es mantingueren restriccions de mobilitat, s'anava a mantenir una venda reduïda de vehicles per particulars. Això va fer que quan van començar a alleujar-se restriccions i confinaments arreu del món, les Fabs reberen peticions arreu del món simultàniament. De normal, aquestes Fabs tenen despeses operatives altes, i treballen amb marges força estrets i amb cues de petició llargues per tal que haja tan poc temps d'inactivitat com siga possible. En rebre les noves peticions, les cues, que ja estaven pressionades, van fer que la indústria automobilística havera d'esperar molts mesos fins poder obtenir els xips que necessitava per poder completar la fabricació dels automòbils.

Això, evidentment és un exemple, però que fa patent la necessitat estratègica d'una indústria de semiconductors "de proximitat", que puga satisfer necessitats més enllà de les de grans corporacions a l'altre costat del món, i centrar-se més en les de la comunitat a la que servixen. Això s'ha manifestat impulsant, per exemple, als EUA i a la UE els *Chips Act* respectius, i a Espanya el *PERTE Chip*.

Aquestes iniciatives s'estan manifestant en construir entorns en els que es fabriquen microxips, però també entorns en els que es plantegen nous dissenys, de tal manera que quan existisca la necessitat de microxips de nou disseny es puguen desenvolupar sense haver de dependre de les cadenes de subministrament i prioritats d'empreses llunyanes. El problema és que en molts casos, la indústria local és limitada en capacitat i no pot assumir les despeses que suposen els acords de Propietat Intel·lectual que els gegants de la indústria demanden i introdueix un altre cas de dependència de les voluntats de corporacions estrangeres. Una solució a aquests problemes és la del Hardware Lliure, de tal manera que equips independents desenvolupen noves tecnologies hardware lliures i els equips puguen col·laborar entre sí. Aquest és un model del que s'ha demostrat la eficàcia, però que en el cas del Hardware seguix sense estar clar si podrà funcionar. Tot i això aquesta és la estratègia que la UE està implementant amb iniciatives com l'*European Processor Initiaitive*.

1.1.4. Computació d'Altes Prestacions

La *Computació d'Altes Prestacions* (HPC) o *supercomputació* és la disciplina de la informàtica en la que s'estudia com integrar l'administració de sistemes i la programació paral·lela amb l'objectiu d'obtenir màquines capaces d'executar aplicacions i resoldre problemes d'una complexitat tal que màquines convencionals no haurien pogut resoldre'ls — o no amb una quantitat de temps raonable. A aquesta disciplina trobem idees d'electrònica digital, arquitectura de computadors, programació de sistemes, algorísmia, entre d'altres camps.

Desde l'invenció dels primers computadors, han hagut problemes, que per la seua extensió, naturalesa, escala o complexitat, queden fora del que una determinada genera-

ció de computadors convencionals pot assolir, però que tot i això necessiten ser resolts. D'aquesta manera, podem imaginar una "finestra de problemes"que a cada generació de computadors pot arribar a resoldre. Evidentment, els computadors moderns són capaços de resoldre problemes increïblement més grans que els que es podia aspirar a resoldre als 80. La idea de la HPC és combinar elements hardware i software de tal manera que es maximitze el rendiment (o l'eficiència) a l'hora de fer càlculs.

Les aplicacions que típicament s'ataquen en HPC són massivament paral·lelitzables com puguen ser simulacions atmosfèriques i climàtiques, simulacions científiques i d'enginyeria de tipus diversos, cerca de recursos naturals, prediccions financeres, entrenament d'Intel·ligències Artificials, entre tantes altres.

Aquesta diversitat d'aplicacions ha portat els governs de les grans potències tecnològiques a fer força per posicionar-se com a capdavanters a la llista TOP500—la llista de supercomputadors més potents—, causant una carrera per construir els supercomputadors més potents, com una demostració de superioritat tecnològica.

Aquestes aplicacions però, poden obtenir beneficis substancials quan fan ús de hardware específicament pensat per elles. Ací no estem parlant d'ASICs, tot i que també podrien ser molt beneficiosos, però per tal de mantenir el propòsit general del que solen disfrutar els supercomputadors, en aquest document no els tractarem en profunditat.

En aquest cas ens referim a processadors que són capaços d'executar moltes instruccions en paral·lel, fent ús de les idees esmentades al punt 1.1.1 amb un alt grau d'eficiència. En aplicacions convencionals, no és típic trobar un nivell de paral·lelisme similar, pel que els processadors enfocats a aplicacions convencionals no tenen la circuiteria necessària per poder aprofitar el paral·lelisme dels supercomputadors.

Existeixen exemples de processadors orientats a supercomputadors, com en puguen ser els *Intel XEON*, o els *AMD EPYC* en els últims anys, tots dos fan ús del joc d'instruccions *x86*, i amb 96 nuclis en el model que més en té. Com hem esmentat al punt 1.1.3, amb l'objectiu de millorar la sobirania tecnològica al camp de HPC, a la UE s'estan dissenyant processadors fent ús de components de codi obert altament personalitzables als objectius que aspira a aconseguir l'usuari final. [1]

1.2 Motivació

Moltes de les companyies que dominen el mercat de la fabricació de semiconductors, incloent la fabricació de memòries, processadors o *sistemes-en-xip* (SOCs), com puguen ser per exemple Intel, IBM, TSMC, AMD, o tantes d'altres, concentren la major part de la propietat intel·lectual i la fan servir per traure rèdit econòmic. Moltes d'aquestes idees i tècniques queden barrades d'accès a la comunitat de recerca pels processos de fabricació i els secretisme industrial. Per més encís, aquest procés s'ha vist accelerat per la Llei de Moore, que ha permès aquestes companyies privilegiades operar amb tècniques, desenvolupar dissenys i alliberar al mercat productes molt superiors al que cap grup d'investigació acadèmic poguera generar.

Aquest procés, però, durant els últims anys s'ha anat frenant, en part per la fi de la Llei de Moore, en arribar al límit físic que permet la miniaturització dels transistors. Això ha causat una reducció en els temps típics a la indústria de disseny i verificació de nous xips. D'una altra banda, la popularitat cada vegada major de ferramentes de codi obert ha ocasionat la creació de grups d'investigació amb l'objectiu d'analitzar i millorar processos de fabricació que prèviament els resultaven inabastables. En aquest sentit, la

[1] Rich Quinnel. *Creating a Custom Processor with RISC-V*. EETimes, 29 de Març del 2019.

iniciativa RISC-V plantejada al 2010 per la *Computer Science Division* de la Universitat de Berkley ha estat una peça clau, que ha impulsat la creació de projectes tant en entorns acadèmics com industrials a col·laborar i generar dissenys que en facen ús. Ara mateix, ja existeixen productes comercials que fan ús de RISC-V, especialment en l'entorn de sistemes encastats, com puguen ser els que ofereix SiFIve.

Certament, els processadors són peces clau en el desenvolupament tecnològic d'un estat i faciliten el desenvolupament de sectors centrals de l'economia. Dit d'una altra manera, construir l'ecosistema i infraestructura necessaris per desenvolupar processadors a un país no és senzill ni barat, però és la solució necessària per assolir un espai tecnològicament estable i segur, evitant els problemes que introdueix la dependència de tercers.

Seguint aquesta línia, s'ha plantejat construir un processador d'alt rendiment com a part del Projecte Lagarto. Aquest projecte es va plantejar al 2010 al *Laboratorio de Microtecnología y Sistemas Embebidos*, al *Centro de Investigación en Computación* del *Instituto Politécnico Nacional* de Mèxic, amb l'objectiu de portar noves idees i innovar al camp de l'arquitectura de computadors. Actualment, el projecte Lagarto col·labora amb el grup d'*Arquitectura de Computadors i Paradigmes Paral·lels* del *Barcelona Supercomputing Center* (BSC). Amb ells, i gràcies al projecte DRAC, es va fabricar el processador Lagarto Hun amb un procés de 65nm. Es tracta d'un processador basat en el joc d'instruccions RISC-V, del qual s'implementen les extensions IMA i inclou 16KiB de memòria cache d'instruccions, i uns altres 16KiB de dades.

Seguidament, es va desenvolupar el processador Lagarto Ka, un processador basat en RISC-V, superescalar de dues vies i fora d'ordre, i amb les extensions IMA. Aquest processador s'està desenvolupant amb l'objectiu d'executar aplicacions d'HPC. A més a més, es busca aprofitar la diversitat i la multidisciplinareitat dels participants del projecte per estimular la fabricació de semiconductors tant a Catalunya (amb l'iniciativa DRAC), com a Mèxic.

En aquest processador, i per tal d'executar una major varietat d'aplicacions HPC, es fa evident la necessitat d'incloure suport per les extensions F i D de RISC-V, que són molt sol·licitades en aplicacions molt diverses del camp, especialment aquelles que simulen el món real. Aquest serà el punt central d'aquest treball i sobre el que ens centrarem als apartats pertinents al desenvolupament de solucions tècniques.

1.3 Objectius

Els objectius que ens hem establert assolir a aquest treball són els següents:

1. Modificacions a l'arquitectura del processador Lagarto Ka:
    - Donar suport a les extensions F i D
    - Afegir suport a la descodificació d'instruccions i establir un estàndard de senyals de control per al flux d'instruccions F i D.
    - Afegir una *unitat de coma flotant* (FPU) al motor d'execució.
    - Integrar la FPU amb el flux d'instruccions que es troba al motor d'execució.
    - Afegir suport per renombrat de registres parametritzat.
    - Modificar la cua d'instruccions d'enters perquè admeta instruccions de coma flotant.
    - Modificar el banc de registres per incloure els registres especificats a les extensions F i D.

2. Verificació de la implementació fent ús de ferramentes d'EDA:
    - Verificar mitjançant simulacions, tant local com asíncronament en un clúster.
    - Emprar FPGAs per simular els circuits implementats.
3. Proposar tècniques microarquitectòniques que permeten incrementar el rendiment:
    - Incloure tècniques de planificació dinàmica per tal d'accelerar l'execució d'instruccions de coma flotant.
    - Incloure tècniques de planificació que permeten amagar latències de l'arquitectura.

1.4 Estructura de la memòria

Aquesta memòria s'estructura en capítols. En cada capítol s'explora un aspecte que ha estat central per al desenvolupament del projecte. En aquest capítol d'introducció es mostra quina és la justificació darrere del desenvolupament del projecte, les circumstàncies del món que l'envolta i quins són els objectius que s'espera assolir.

El capítol segon detalla els estàndards sobre els que es treballarà durant aquest projecte i s'aporta informació que serà rellevant per al lector per tal d'entendre com funcionen determinats aspectes del sistema especificat.

El tercer capítol esmenta quins han estat els requisits concrets que han condicionat el desenvolupament del projecte i quin és el punt de partida desde el que es començarà a treballar.

En el quart capítol es desenvolupa una exploració primer a nivell arquitectònic de com funcionarà la ruta de dades que s'implementarà i després s'entra en els detalls microarquitectònics de cada component pel que passarà la ruta de dades.

El cinqué capítol mostra quin ha estat el procés de verificació, les fases per les que ha passat per validar el disseny implementat i s'exposen els resultats de rendiment que finalment s'han assolit.

Finalment, a la conclusió es repassen els objectius assolits, es fa una crítica al procés i a l'estat de l'art i s'exploren futures vies de desenvolupament.

CAPÍTOL 2
Estat de l'Art

2.1 El Joc d'Instruccions RISC-V

El joc d'instruccions RISC-V és una arquitectura de conjunt d'instruccions dissenyada per a processadors. *RISC* significa *Reduced Instruction Set Computer* (ordinador amb joc d'instruccions reduït), el que indica que RISC-V té un nombre limitat d'instruccions bàsiques, però altament optimitzades. Aquesta arquitectura és lliure i de codi obert, el que significa que els dissenyadors de processadors poden utilitzar-la sense restriccions i modificar-la segons les seves necessitats.

El disseny RISC-V ofereix una base senzilla i elegant per a processadors. Les seves instruccions estan dissenyades per ser clares, fàcils d'entendre i implementar. RISC-V és modular i permet l'extensió amb conjunts d'instruccions addicionals segons les necessitats específiques de l'aplicació. Això fa que siga una arquitectura molt versàtil, que pot ser utilitzada en diferents tipus de dispositius, des de microcontroladors xicotets fins a grans servidors.

Un altre avantatge significatiu de RISC-V és la seva naturalesa de codi obert. Aquesta arquitectura ha guanyat una gran comunitat de desenvolupadors i entusiastes que col·laboren per a millorar i expandir les seues característiques. Hi ha una àmplia gamma de recursos, eines de desenvolupament i suport disponibles per a treballar amb RISC-V, el que facilita la implementació i el desenvolupament de programari compatible.

RISC-V s'ha convertit en una opció popular en diverses àrees. És utilitzat en la investigació acadèmica, ja que ofereix una plataforma flexible per a realitzar experiments i desenvolupar nous algoritmes. També és utilitzat en el desenvolupament de processadors personalitzats, on els dissenyadors poden adaptar l'arquitectura per a satisfer les necessitats específiques d'una aplicació o sistema. A més, RISC-V té aplicacions en el camp de la robòtica, els microcontroladors i altres àmbits on es requereix un disseny eficient i un control precís.

A l'estàndard de RISC-V es defineixen una sèrie d'extensions que incrementen les funcionalitats bàsiques d'un processador i que es poden identificar a la nomenclatura seguint la convenció que s'estableix a la taula 2.1 .

D'aquestes extensions, el processador Lagarto Ka implementarà la *RV64GV* (*RV64IMAFDVZicsr_Zifencei*), però a aquest treball ens centrarem en el procés d'implementació de les extensions F i D, i com s'han inclòs a processador RV64IMA.

Subset	Name	Implies
Base ISA		
Integer	I	
Reduced Integer	E	
Standard Unprivileged Extensions		
Integer Multiplication and Division	M	
Atomics	A	
Single-Precision Floating-Point	F	Zicsr
Double-Precision Floating-Point	D	F
General	G	IMADZifencei
Quad-Precision Floating-Point	Q	D
16-bit Compressed Instructions	C	
Bit Manipulation	B	
Cryptography Extensions	K	
Dynamic Languages	J	
Packed-SIMD Extensions	P	
Vector Extensions	V	
Control and Status Register Access	Zicsr	
Instruction-Fetch Fence	Zifence	
Misaligned Atomics	Zam	A
Total Store Ordering	Ztso	
Standard Supervisor-Level Extensions		
Supervisor-level extension "def"	Sdef	
Standard Hypervisor-Level Extensions		
Hypervisor-level extension "ghi"	Hghi	
Standard Machine-Level Extensions		
Machine-level extension "jkl"	Zxmjkl	
Non-Standard Extensions		
Non-standard extension "mno"	Xmno	

Taula 2.1: Estàndard actual per la nomenclatura d'extensions RISC-V

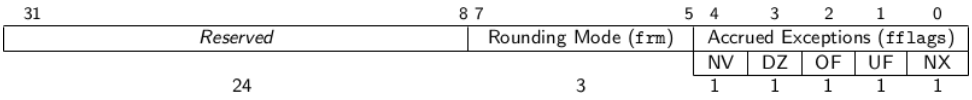


Figura 2.1: Registre de control i estat per flotants

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards ∞)
100	RMM	Round to Nearest, ties to Max Magnitude
101		*Reserved for future use.*
110		*Reserved for future use.*
111	DYN	In the instruction's rm field, selects dynamic rounding mode; In Rounding Mode register, *reserved*.

Taula 2.2: Codificació de modes d'arredoniment

2.1.1. L'extensió F de RISC-V

Ací descriurem les instruccions que afig l'extensió F de RISC-V, que inclou operacions de càlcul per nombres de simple precisió seguint l'estàndard *IEEE-754-2008* [1]. L'extensió F depèn de l'extensió *Zicsr*.

L'extensió afig 32 registres lògics anomenats `f0-f31`, cadascun de 32 bits d'ample i un registre de control i estat per flotants, `fcsr`. La majoria d'instruccions de l'extensió F operen sobre valors en els registres `f`. Les instruccions de `load` i `store` per flotants mouen valors entre els registres i memòria. També s'inclouen instruccions per moure valors entre els registres d'enters i els de flotants.

Nous accessos CSR

El registre `fcsr` és un registre de 32 bits amb les dades estructurades com s'indica a la figura 2.1.

Aquest registre es pot llegir amb la pseudoinstrucció `FRCSR` i escriure amb `FSCSR`. La primera copia el valor de `fcsr` al registre `rd`; la segona copia `fcsr` a `rd` i es sobreescriu amb el valor de `rs1`.

Es permet l'accés individual als camps que s'especifiquen a la figura 2.1 amb pseudoinstruccions. La instrucció `FRRM` llig el valor del camp *Rounding Mode* `frm` i el copia als tres bits menys significatius del registre `rd`, i ompli la resta amb zeros. `FSRM` copia el valor original de `frm` a `rd`, i es sobreescriu el camp `frm` amb els tres bits menys significatius de `rs1`. També es defineixen de manera anàloga les instruccions `FRFLAGS` i `FSFLAGS` per al camp de `fflags`.

Les operacions de coma flotant fan ús d'un mode estàtic d'arredoniment especificat en la instrucció, o bé fan ús d'un mode dinàmic en el que es pren el mode d'arredoniment especificat en `frm`. Els modes d'arredoniment dels que fa ús RISC-V queden reflectits a la taula 2.2. Per al cas del valor 111, al camp `rm` d'una instrucció, es triarà el mode dinàmic especificat a `frm`.

Les excepcions que ocorren per qualsevol instrucció aritmètica de coma flotant queden emmagatzemades al camp `fflags` seguint la relació especificada a la taula 2.3. El joc

Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

Taula 2.3: Codificació dels tipus d'excepcions

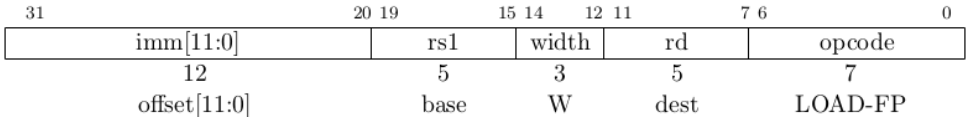


Figura 2.2: Format de les Instruccions FP-Load

d'instruccions base de RISC-V no suporta generar *traps* en activar qualsevol *flag*, en el seu lloc, aquestes activacions s'han de comprovar per programa.

NaN Canònica

Una qüestió important a tenir en compte quan es parla de coma flotant és la de NaN. Aquest és un punt en el que RISC-V F es desvia de l'estàndard d'IEEE-754. A l'estàndard d'IEEE, al punt 6.2.3 sobre propagació de NaNs s'indica que quan un dels operands té un valor NaN com a entrada, el contingut d'aquest NaN s'hauria de propagar al resultat, i quan en té més d'un que té un valor NaN s'hauria de propagar el contingut d'un dels NaNs.[1]. Això es justifica a l'estàndard com una decisió de disseny amb la idea de facilitar la traçabilitat d'errors.

Això però a l'estàndard RISC-V no es així, i en el seu lloc apareix la figura de NaN canònic. El NaN canònic de RISC-V és un valor NaN amb signe positiu, i tots els bits de la mantissa a 0 excepte el primer que està a 1. Per a valors flotants de precisió simple el patró de 32 bits seria `0x7fc00000`.

Aquest valor serà el resultat de qualsevol operació que tinga per resultat un NaN. Això vol dir que els moviments de dades entre registres, o entre memòria i un registre han de canonitzar NaNs. De la mateixa manera, tot resultat d'una operació aritmètica que convencionalment seria NaN, com per exemple $\sqrt{-1}$, haurà de retornar una NaN canònica, i no qualsevol valor NaN. Aquesta decisió es justifica a l'estàndard amb una esperada reducció de cost hardware.

Operacions d'Accés a Memòria

Les operacions d'accés a Memòria són les clàssiques LOAD i STORE, però que enlloc d'emmagatzemar el valor arriba o llegir valor que s'envia a Memòria de registres lògics d'enters es llig de registres per Flotants. El format que fan servir les instruccions de LOAD i STORE es el que es presenta a les figures 2.2 i 2.3 respectivament.

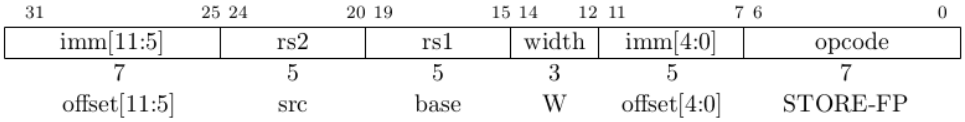


Figura 2.3: Format de les Instruccions de FP-Store

Camp `fmt`	Mnemònic	Significat
00	S	Precisió simple de 32 bits
01	D	Precisió doble de 64 bits
10	H	Precisió mitja de 16 bits
11	Q	Precisió quàdruple de 128 bits

Taula 2.4: Codificacions del format de coma flotant al camp fmt.

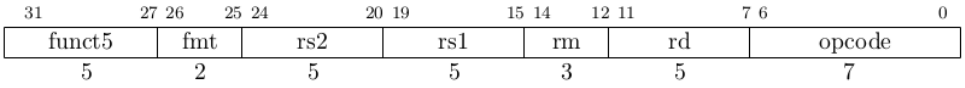


Figura 2.4: Format general de les operacions aritmètiques de coma flotant

En les instruccions de load, el registre d'enters *rs1* indica un registre que conté una adreça base sobre la que es sumarà el valor immediat de 12 bits *imm*. Aquesta serà l'adreça que es llegirà en memòria. El valor que retorne memòria s'emmagatzemarà en el registre lògic per flotants *rd*.

En les instruccions d'store l'adreça a la que dirigir-se a memòria es calcula de manera anàloga a les operacions de load, fent servir el registre d'enters *rs1* i sumant-li l'immediat *imm* de 12 bits, però en aquest cas es llegirà el valor contingut en el registre *rs2* i s'escriurà a l'adreça de memòria especificada.

Aquestes operacions hauran de ser atòmiques si l'adreça a la que accedeixen està alineada; també cal comentar que aquestes operacions no canonitzaran NaNs i es limiten a efectuar moviments de dades sense alterar els bits que mouen.

Operacions Aritmètiques de FP

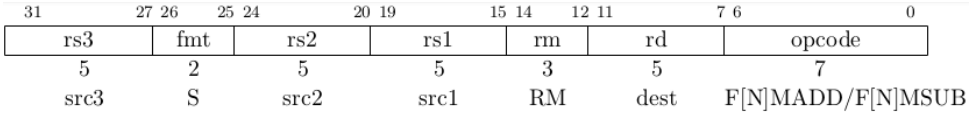
Les operacions aritmètiques amb un o dos operands font fan ús del format estàndard de RISC-V per instruccions R amb el codi d'operació OP-FP. Les operacions FADD.S i FMULT.S efectuen la suma i multiplicació dels valors de coma flotant de precisió simple entre *rs1* i *rs2* respectivament. Les operacions FSUB.S i FDIV.s realitzen la resta i divisió en coma flotant de precisió simple d'*rs1* d'*rs2*. FSQRT.S calcula l'arrel quadrada del valor d'*rs1*. Totes aquestes operacions guarden el resultat en el registre *rd*.

El valor de 2 bits *fmt* codifica el format de coma flotant que s'especifica a la taula 2.4. Totes les operacions que son susceptibles d'arredoniments han de seleccionar un valor de la taula 2.2.

Les operacions de FMIN.S i FMAX.S escriuen respectivament el valor més petit o més gran respectivament de la parella de valors que es reben dels registres *rs1* i *rs2* al registre *rd*. Per aquestes operacions i sols per aquestes dues el valor de FP -0.0 és menor que +0.0. Si els dos valors font són NaNs, el resultat és una NaN canònica. Si sols un dels valors és NaN, el resultat és el valor que no és NaN. Si cap dels dos operands és NaN, s'activarà la marca d'excepció de *operació invàlida*, inclús quan el resultat no siga NaN.

Les operacions *fused* de multiplicació-suma fan ús d'un nou format estàndard per instruccions: l'R4. En aquest format es presenten tres registres font (*rs1*, *rs2* i *rs3*) i un registre destí (*rd*). El podeu veure a la figura 2.5.

FMADD.S multiplica els dos valors *rs1* i *rs2* i els suma el tercer valor *rs3*, i escriu el resultat final a *rd*. FMADD.S efectua la següent operació: $rd = rs1 \times rs2 + rs3$. De manera similar, FMSUB.S efectua la següent operació: $rd = rs1 \times rs2 - rs3$.



31–27	26–25	24–20	19–15	14–12	11–7	6–0
rs3	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
src3	S	src2	src1	RM	dest	F[N]MADD/F[N]MSUB

Figura 2.5: Format de les operacions *fused*

31–27	26–25	24–20	19–15	14–12	11–7	6–0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.*int.fmt*	S	W[U]/L[U]	src	RM	dest	OP-FP
FCVT.*fmt.int*	S	W[U]/L[U]	src	RM	dest	OP-FP

Figura 2.6: Estructura d'una operació de conversió

FNMSUB.S multiplica els dos valors *rs1* i *rs2*, al resultat li canvia el signe i li suma el valor *rs3* i escriu el resultat final a *rd*. FNMSUB.S efectua la següent operació $rd = -(rs1 \times rs2) + rs3$. De la mareixa manera, FNMADD.S efectua l'operació $rd = -(rs1 \times rs2) - rs3$. (No es una errata, el primer *rs3* suma i el segon resta)

Les operacions *fused* hauran d'activar la marca d'excepció d'*operació invàlida* quan la multiplicació siga una de $\infty \times 0$. Es marcarà inclús quan la operació indique la suma d'un NaN al final ($\infty \times 0 + NaN$), cosa a la que l'estàndard IEEE-754 no obliga.

Operacions de Conversió, Injecció de Signe i Moviment

Les operacions de conversió d'enters a FP o d'FP a enter es codifiquen fent servir el codi d'operació OP-FP. FCVT.W.S i FCVT.L.S converteixen valors de precisió simple de coma flotant provinents del registre *rs1* a valors d'enters amb signe de 32 o 64 bits respectivament que es guarden al registre *rd*. Les operacions FCVT.S.W i FCVT.S.L converteixen valors d'enters amb signe de 32 o 64 bits respectivament que venen del registre *rs1* en valors de coma flotant de simple precisió que es guarden al registre *rd*. Per cadascuna d'aquestes operacions existeix una variant que pren el valor d'enters com a sense signe: FCVT.WU.S, FCVT.LU.S, FCVT.S.WU i FCVT.S.LU. Les operacions FCVT.W[U].S extenen el signe a la resta del registre quan la longitud de les dades és major que 32 bits. Les operacions FCVT.L[U].S i FCVT.S.L[U] s'hauran d'incloure sols per processadors que donen suport al joc d'instruccions de 64 bits. Si el resultat arredonit no es pot representar en el format de destí, se'l fixa al valor més pròxim i s'activa la marca d'*operació invàlida*. El valor infinit negatiu es fixarà al valor més baix que puga suportar cada format de destí; els valors de infinit i NaN es fixaran al valor més al possible per al format de destí.

Les operacions de conversió activaran la marca d'*Inexacte* quan el resultat de l'operació no represente el mateix valor que el del registre font i no haja activat la marca d'*operació invàlida*.

Les instruccions de FP a FP d'injecció de signe FSGNJ.S, FSGNJN.S i FSGNJX.S produeixen un resultat que replica tots els bits del registre font *rs1* excepte el de signe. Per FSGNJ.S el bit de signe de resultat és el de *rs2*; per FSGNJN.S és el contrari del de *rs2*; per FSGNJX.S és el bit que resulta de fer XOR als bits de signe d'*rs1* i *rs2*. Aquestes operacions no canonitzen NaNs. L'estàndard afig una sèrie de pseudoinstruccions que fan ús d'aquestes instruccions per facilitar-li la feina al programador: *FMV.S ry, rx* mou el valor de *rx* a *ry*, FNEG.S ry, rx deixa en *ry* el valor oposat a *rx*, *FABS.S ry, rx* deixa a *ry* el valor absolut d'*rx*. Aquestes pseudoinstruccions fan ús de les instruccions FSGNJ.S ry, rx, rx, FSGNJN.S ry, rx, rx i FSGNX.S ry, rx, rx respectivament.

A l'extensió F de RISC-V també s'afigen instruccions per moure patrons de bits entre registres d'enters i de FP. FMV.X.W mou el patró de bits emmagatzemat al registre de

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FMV.X.W	S	0	src	000	dest	OP-FP
FMV.W.X	S	0	src	000	dest	OP-FP

Figura 2.7: Estructura de les instruccions de FMV.X.W

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCMP	S	src2	src1	EQ/LT/LE	dest	OP-FP

Figura 2.8: Estructura de les instruccions de comparació per FP.

coma flotant *rs1* i el guarda als 32 LSB del registre d'enters *rd*. Els bits no es modifiquen, ni tampoc s'altera el contingut de les NaNs, el que si que canvia és els 32 MSB que cadascun d'ells pren el valor del bit de signe.

FMV.W.X mou el patró dels 32 LSB del registre d'enters font *rs1* i el deixa al registre de coma flotant *rd*. En aquesta operació tampoc es modificaran els patrons de les NaN.

Operacions de Comparació i Classificació

Les instrucció de comparació per coma flotant són FEQ.S, FLT.S i FLE.S efecuen les comparacions entre registres $rs1 = rs2$, $rs1 < rs2$ i $rs1 \leq rs2$ respectivament i escriuen el valor 1 al LSB del registre d'enters *rd* quan la condició es compleix i 0 sinó.

FLT.S i FLE.S marquen l'excepció de *operació invàlida* si cap dels valors d'entrada es NaN. FEQ.S marca l'excepció de *operació invàlida* amb entrada de NaN ha marcat una excepció. Per les 3 comparacions el resultat serà 0 si cap dels operands és NaN.

La instrucció FCLASS.S examina el valor del registre de coma flotant *rs1* i escriu al registre d'enters *rd* una màscara de 10 bits que indica la classe de nombre de FP. El format de la màscara es descriu a la taula 2.5. Tots els altres bits del registre es marquen a 0. Dels bits de la màscara sols 1 s'activarà, la resta quedaran desactivats. L'operació de classificació no activa marques d'excepció.

Índex del bit	Significat
0	*rs1* és $-\infty$
1	*rs1* és un nombre negatiu normal
2	*rs1* és un nombre negatiu subnormal
3	*rs1* és -0.0
4	*rs1* és $+0.0$
5	*rs1* és un nombre positiu subnormal
6	*rs1* és un nombre positiu normal
7	*rs1* és $+\infty$
8	*rs1* és un NaN que ha marcat una excepció
9	*rs1* és un NaN que no ha marcat excepció

Taula 2.5: Format de la màscara d'una operació de classificació

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCLASS	S	0	src	001	dest	OP-FP

Figura 2.9: Estructura de les instruccions de classificació

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.S.D	S	D	src	RM	dest	OP-FP
FCVT.D.S	D	S	src	RM	dest	OP-FP

Figura 2.10: Estructura de les instruccions de conversió entre formats de FP

2.1.2. L'extensió D de RISC-V

L'extensió D de RISC-V afig un conjunt d'operacions per operacions sobre valors de doble precisió d'acord a l'estàndard a l'estàdard IEEE 754-2008 [1]. El nom de *D* ve de que es tracten valors de *doble* precisió.

El primer que s'ha de tractar sobre l'extensió D és que amplia la longitud dels registres de coma flotant de 32 bits a 64. Cadascún d'aquests registres pot guardar valors de 32 o 64 bits.

NaN-Boxing

Els valors que tenen una extensió menor de 64 bits es guardaran als LSB del registre. Els bits que queden fora de l'extensió del valor es quedaran com a 1s. Per això els valors d'aquestes longituds en llegir-los amb l'extensió completa del registre s'interpretaran com a valors NaN negatius que no han marcat cap excepció.

Les operacions de transferècia de dades mouen patrons de bits cap a o desde registres de FP (FLn, FSn, FMV.X.n i FMV.n.X). Quan es produeix una transferència de bits als registres de FP i la seua longitud és menor que la dels registres, se li aplicarà NaN-Boxing; per les transferències que surten dels registres de FP es prendran els n LSB.

A més a més totes les operacions de FP aplicaran NaN-Boxing per les instruccions que especifiquen un format de sortida menor que el que indica la longitud del registre. Si una operació pren valors amb una longitud menor que la del registre i el valor que rep no està NaN-Boxed, s'interpretarà com si es tractara d'una NaN canònica.

Operacions

Fora d'aquesta excepció, es defineixen instruccions que actuen de manera anàloga a la descrita a l'extensió F, però operant amb valors de 64 bits amb el format de doble precisió del IEEE 754-2008 [1]. En la seua estructura canvien al camp de *Width* (W) (per les operacions de memòria) i el camp de *Format* (fmt) (per la resta d'operacions) de 3 bits el valor per simple precisió (S) pel de doble precisió (D).

Addicionalment, s'afigen operacions de conversió de simple precisió a doble precisió i de doble precisió a simple precisió(FCVT.S.D i FCVT.D.S). Tant el registre font *rs1* com el de destí *rd* apunten a registres de FP. El camp *rs2* indica el format del registre font i el camp *fmt* indica el format del registre destí. FCVT.S.D arredoneix seguint la norma especificada al camp rm, mentre que FCVT.D.S no arredoneix mai. Podeu veure l'estructura de les seues instruccions a la figura 2.10.

Nom	Nom Comú	Base	Dígits de Mantissa	Bits d'exponent	Biaix d'exponent	Valor màxim
Binary16	Mitja Precisió	2	11	5	15	2^{15}
Binary32	Precisió Simple	2	24	8	127	2^{127}
Binary64	Precisió Doble	2	53	11	1023	2^{1023}
Binary128	Precisió Quàdruple	2	113	15	16383	2^{16383}
Binary256	Precisió Octuple	2	237	19	262143	2^{262143}
Decimal32		10	7	7.58	101	2^{96}
Decimal64		10	16	9.58	398	2^{384}
Decimal128		10	34	13.58	6176	2^{6144}

Taula 2.6: Formats bàsics de l'estàndard IEEE-754

2.2 L'estàndard sobre valors decimals: IEEE-754

L'estàndard IEEE-754 es un estàndard tècnic per aritmètica de coma flotant establert el 1985 per l'Institut d'Enginyers Elèctrics i Electrònics. L'objectiu de l'estàndard era unificar la multiplicitat de formats de coma flotant que existien a l'època, cosa que feia que els valors guardats en aquests formats no fóren portables ni fiables. Hui en dia la majoria d'FPUs implementen l'estàndard IEEE-754.

Formats i Valors Vàlids

Aquest estàndard defineix formats aritmètics, formats d'intercanvi, regles d'arredoniment, operacions i control d'excepcions.

Pel que fa als formats, l'estàndard permet definir formats de longitud arbitrària fent sevir els paràmetres b (base, típicament 2 per representacions en binari o 10 per representacions decimals), p (precisió, la longitud de la mantissa a representar) i un rang d'exponents tal que emin = 1 - emax, és a dir, que l'exponent quede centrat sobre l'1.

El format no permet incloure valors amb infinita precisió, i es limita a valors que es puguen representar amb els següents paràmetres: signe s (1 ó 0 per negatiu o positiu respectivament), coefincient c que no pot tenir més de p dígits al representar-lo en la base b i un exponent q tal que $emin \leq q + p - 1 \leq emax$. Un valor amb paràmetres s, c i q que s'adapte als requeriments i que tinga com a paràmetres b, p, emin i emax representarà el valor donat per la fòrmula $(-1)^s \times c \times b^q$.

També es defineixen altres valors com dos zeros (-0.0 i +0.0), dos infinits ($-\infty$ i $+\infty$) i dos tipus de *not-a-number*, NaN (qNaN o *quiet* NaN, i sNaN o *signaling* NaN). Addicionalment, quan el valor de l'exponent és el mínim possible es passa a una representació especial per a nombres subnormals.

Tot i la multiplicitat de possibles codificacions que ofereix l'estàndard, se'n recomanen uns quants que tenen una gran adopció pública per facilitar l'intercanvi d'informació. Aquests formats venen descrits en la taula 2.6.

Arredoniment

Quan el resultat d'una operació no es pot representar en l'estàndard IEEE-754, aleshores es produiràun arredoniment a la xifra menys significativa de la mantissa. Aquests arredoniments es defineixen en 5 tipus distints:

- Arredoniment al més pròxim, empats a parells. S'arredoneix al valor més pròxim, en cas que el nombre quede al bell mig s'arredoneix de tal manera que el dígit menys significatiu de la mantissa siga parell.
- Arredoniment al més pròxim, empats s'allunyen de zero. S'arredoneix al valor més pròxim, en cas que el nombre quede al bell mig s'arredoneix al següent valor per damunt, per als positius, o per baix, per als negatius.
- Arredoniment cap a 0. S'arredoneix sempre cap a zero, típicament se li diu *truncar*.
- Arredoniment cap a $+\infty$. També se li diu arredonir cap amunt o funció *sostre*.
- Arredoniment cap a $-\infty$. També se li diu arredonir cap avall o funció *sòl*.

Operacions Necessàries

L'estàndard també defineix una sèrie d'operacions que s'han d'implementar necessàriament:

- Conversions desde i cap a enters.
- Valor previ i següent.
- Operacions Aritmètiques (suma, resta, multiplicació, divisió, arrel quadrada, operació axpy, sobrant d'una divisió, mínim i màxim).
- Conversió entre formats i de i a cadenes de caracters.
- Còpia i manipulació de signe.
- Comparacions
- Classificació dels nombres i comprovació de NaNs.
- Comprovació i activació de les marques d'excepció

Excepcions

També es defineixen una sèrie de marques mínimes d'excepció que s'han d'activar en circumstàncies especials. Aquestes marques són les mateixes que defineix l'estàndard RISC-V. Si mirem en detall quina és la definció de cadascuna d'aquestes marques trobem el següent:

- Operació Invàlida: Típicament no definit matemàticament, per exemple $\sqrt{-1}$. Activa la marca però no llança cap excepció.
- Divisió entre zero: Una operació entre operands finits retorna un valor infinit, per exemple $1/0$. Per defecte retorna $\pm\infty$.
- Overflow: Un resultat finit és massa gran per representar-lo adequadament (l'exponent queda fora del rang permés). Per defecte retorna $\pm\infty$
- Underflow: Un resultat és molt petit (fora del rang normal). Per defecte es retorna un nombre menor o igual que el nombre normal amb la magnitud més petita possible. Un nombre subnormal sempre implica una excepció d'underflow, però si el resultat és exacte no activa la marca.
- Inexacte: El resultat exacte no és respresentable. Per defecte, es retorna el valor correctament arredonit.

CAPÍTOL 3
Anàlisi del Problema

En aquesta secció esmentarem els requisit que necessita el desenvolupament, comentarem la estructura del processador Lagarto Ka abans de l'inici del projecte, es comentaran possibles implementacions, tot i que l'exploració en detall de les solucions escollides es desenvoluparà en posteriors capítols.

Finalment, també parlarem d'aspectes legals, tant de qüestions de propietat intel·lectual, com de llicències.

3.1 Requisits del projecte

Dins del desenvolupament del projecte, podem observar diversos requisits que condicionaran les implementacions escollides:

- Implementar una ruta de dades que s'adherisca a les especificacions de l'estàndard de RISC-V.
- Permetre la flexibilitat necessària per mantenir la resta d'extensions ja implementades en el processador Lagarto Ka, i l'addició de noves.
- Fer ús de l'FPU de codi obert, `FPnew`, desenvolupada per l'OpenHW Group.
- Implementar l'RTL amb SystemVerilog, adherint-se a les regles d'estil que descriuen la resta del processador.
- Construir una implementació parametritzada.
- Fer ús de dues FPUs.
- Mantenir una segmentació suficient que permeta obtenir una freqüència de rellotge alta.
- Minimitzar l'ús d'àrea per tal de no excedir els límits especificats.
- Mantenir un consum energètic baix i eficient.
- Implementar la ruta de dades de tal manera que maximitze l'IPC.
- Verificar el correcte funcionament lògic de la nova implementació, tant en les instruccions de les extensions F i D, com el funcionament correcte de tot el conjunt.
- Limitar-se a fer ús de ferramentes de disseny i verificació lliures, o de les que es disposa de llicència.

- Acabar el disseny i implementació abans de l'1 de maig del 2023 i la verificació abans de l'1 de juny del mateix any.

3.2 Implementació prèvia

En aquest capítol farem una ullada a la implementació prèvia al desenvolupament del projecte, per tal d'esbrinar quins components es veuran afectats.

Recordem que Lagarto Ka serà un processador RV64G, això vol dir que ha d'implementar les extensions IMAFDZifencei_Zicsr.

També és important recordar que Lagarto Ka és un processador segmentat i superescalar de 2 vies amb execució OoO. Les fases que implementa el processador són les que es mostren a la figura 3.1.

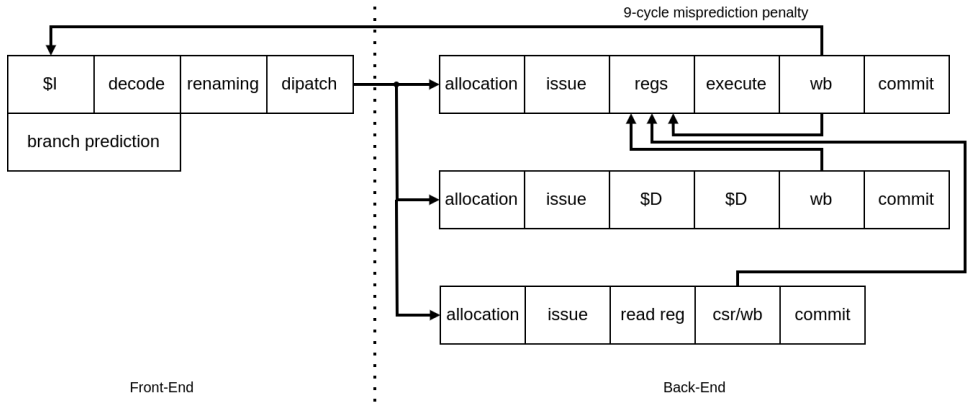


Figura 3.1: Fases d'una instrucció al processador Lagarto Ka.[3]

Aquestes fases es disposen en una sèrie de components lògics. Aquests components són els que es mostren a la figura 3.2.

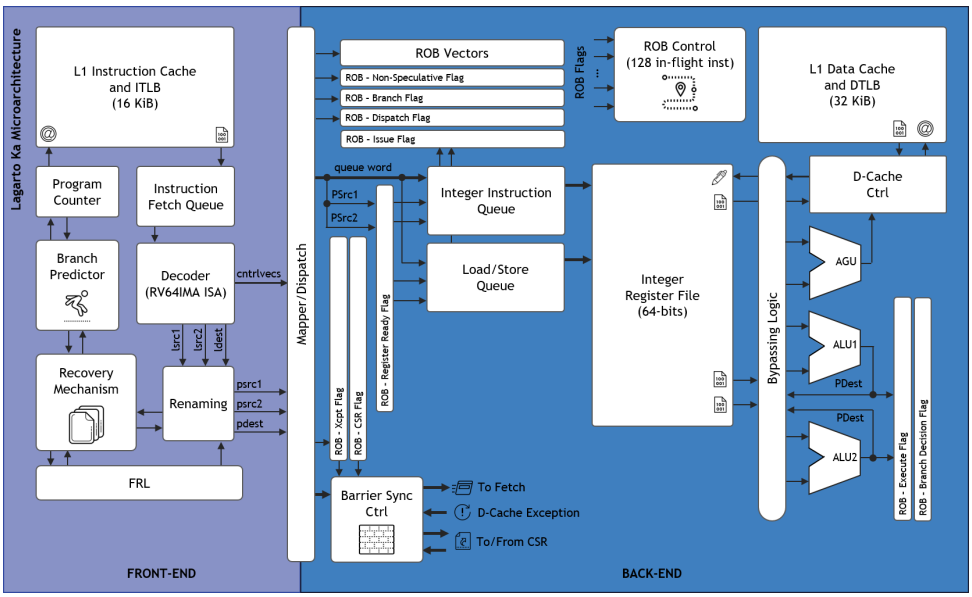


Figura 3.2: Vista General de la microarquitectura Lagarto Ka

Pel que fa al diagrama de fases, es pot observar que la predicció de salts es produeix en paral·lel a l'accés a la cache d'instruccions. Aquest accés ocupa dos cicles. D'altra ban-

da, l'accés a la cache de dades ocupa tres cicles. Un altre aspecte cridaner és la separació en dos blocs ben delimitats a la ruta de dades: la que ocorre en ordre i la que ocorre OoO; l'element que s'encarrega de trencar l'ordre són les cues d'enters (IQ), i la cua de *Load/Store* (LSQ), i si bé les fases d'assignació (*allocation*) i de despertar (*wake-up*) apareixen com a adjacents al diagrama, això és una representació optimista, i poden estar separades per un nombre indeterminat de cicles mentre esperen a rebre les dades que necessiten per dur a terme les seues respectives operacions.

Pel que fa a la ruta d'enters, la fase d'execució es marca com a un únic cicle. Això és així per totes les operacions d'enters excepte les de divisió, que prenen 3 cicles.

Notem també que el processador té dues ALUs per fer càlculs d'enters, però sols una AGU, que és la que permet els accessos a memòria, això fa que sols es permeta que comence un accés a memòria a cada cicle, i que comencen dues operacions d'enters. Tot i això es limita el llançament d'operacions a dues per cicle (siguen LS o aritmètiques), de tal manera que en un únic cicle completen (*Commit*) com a màxim dues operacions.

Si observem amb detall, podem veure el curtcircuit entre les dues fases que fan ús del banc de registres: la de lectura i la d'escriptura. Sense aquest curtcircuit, les operacions desperten el cicle en el que s'escriuen al registre, i tarden dos cicles més fins llegir la dada al banc; per contra, aquest curtcircuit permet amagar aquests dos cicles de latència però necessita una notificació de disponibilitat de dades a la IQ dos cicles abans que l'operació complete, de tal manera que l'instrucció concreta desperte i puga rebre les dades quan arribe a la fase de lectura de registres.

Per les operacions d'enters això no és cap problema i pot funcionar amb naturalitat, però quan existeix una dependència amb operacions de *load*, si aquesta *load* espera que una dada estiga en cache però no està, es produirà un despertar especulatiu fallat, i s'haurà de retornar a la cua.

En general, el flux d'operacions està controlat per un *Reorder Buffer* distribuït, això implica que han d'haver molts més llocs on emmagatzemar informació de control. Això a priori podria semblar negatiu, però la idea darrere d'aquesta implementació és que d'aquesta manera, a l'hora de fer el disseny físic es minimitza la quantitat de cables que han de travessar tot el processador, ja que la informació de control està emmagatzemada prop d'on es fa servir. D'aquesta manera es redueix l'ús d'àrea (i de manera indirecta el consum elèctric) i es pot incrementar la freqüència de rellotge (ja que els camins crítics queden físicament més curts).

Aquesta aproximació, però, necessita que els components tinguen protocols de control distribuïts, enlloc d'una unitat de control centralitzada que dicte el comportament i interacció, i mantinga l'estat entre els diversos elements.

Val a dir que al diagrama de fases, no estan contemplades operacions que afecten al *Control and Status Register* (CSR), ni les operacions de *Fence*.

Sobre les *Fence* no entrarem tampoc en massa detall, però comentarem que aturen l'execució segmentada, de tal manera que es garanteix l'ordre d'accessos a memòria forçant que no puguen entrar noves instruccions fins que no complete el *Fence*, i per tant, les anteriors instruccions.

Pel que fa a les operacions de CSR, no estan al diagrama de cicles que es mostra a la figura 3.1, tot i això se li dona suport. Aquestes operacions, per la seua poca freqüència en aplicacions típiques, es duen a terme en ordre. El bloc lògic que guarda les dades del CSR està fora del processador Lagarto Ka, i es comunica a través d'una interfície dedicada.

3.3 Decisions d'Implementació

Els components microarquitectònics que s'hauran de modificar o implementar com a nous són els següents:

- Decodificador
- *Mapper*
- Banc de Registres
- Renombrat
- Cua de Flotants
- Acomodar l'FPU
- Lògica de Bypass
- Mecanisme de Recuperació

Els canvis a implementar en cadascun d'aquests components seran més grans o més petits, però la idea es que aquesta siga una llista exhaustiva.

Pel que fa al Decodificador, s'ha d'interpretar la instrucció tal qual s'emmagatzema en memòria i transformar-la en els senyals adequats de control interns adequats per dins del processador. En la implementació de Lagarto Ka, tenim una memòria amb cadascuna de les instruccions que suporta el processador com a adreça que emmagatzema els valors lògics dels senyals de control. En aquest cas, per ampliar les funcionalitats.

Al *Mapper* caldrà afegir la funcionalitat per construir els vectors de control que s'enviaran a la cua de flotants per tal que es duga a terme l'instrucció concreta adequadament. Ací haurem de determinar quins senyals de control s'hauran d'enviar cap al back-end (la zona OoO), quines dades no són necessàries i quin format hauran de tindre per tal que siguen fàcils d'interpretar i moure per la ruta de dades.

Ací ens resultarà beneficiós que el ROB siga distribuït, ja que podrem afegir noves funcionalitats sense haver de tocar una estructura de control centralitzada.

Pel que fa al Banc de Registres se'ns presenta una disjuntiva: l'extensió F de RISC-V demana afegir 32 registres lògics de dades nous, a més dels 32 que inclou l'ISA de base. La qüestió és que podríem ampliar el banc de registres ja existent, o bé afegir-ne un de nou, i afegir la lògica que fora necessària per discriminar entre els dos. A priori semblaria més senzill crear-ne un banc nou, perquè no cal modificar elements que ja funcionen correctament, i sols caldria validar la lògica que els discrimina. El problema és que a l'extensió F existeixen instruccions que mouen dades de registres d'enters a flotants, i a l'inversa, pel que no és tan senzill com detectar una instrucció de flotants i descartar el banc d'enters.

Addicionalment, si recordem la descripció del processador és un superescalar de dues vies. El banc de registres existent està pensat per operar sobre dues operacions, o siga que accepta l'escriptura de 3 registres, i la lectura de 6 en un mateix cicle. Crear-ne un nou implicaria efectivament tenir 6 vies d'escriptura i 14 de lectura. Però en el millor dels casos sols es farien servir 2 vies d'escriptura i 6 de lectura. Tot això té un sobrecost en àrea que preferim no assumir, i en el seu lloc, decidim modificar el banc de registres per tal que tinga 64 registres lògics, i 128 de físics.

Aquests canvis però també s'hauran de traduir al dispositiu de renombrat, que sols està pensat per treballar sobre 32 registres lògics. Cal esmentar que a l'extensió F es

necessita poder llegir 3 registres font per les instruccions *fused*, això ens força a afegir la funcionalitat necessària per poder sol·licitar fins a 6 registres font a la *Register Alias Table* (RAT). Això però suposa un canvi menor, ja que el mecanisme de renombrat està parametritzat, pel que afegir uns quants registres més i uns quants ports més no hauria de ser massa costós en temps del dissenyador; tampoc ho hauria de ser en àrea.

La cua de flotants és una estructura complexa, ja que ha de poder emmagatzemar informació de control i les dades de les instruccions; ha de guardar quins registres han fet WB, ha d'alliberar instruccions que facen *Commit*, ha de poder-se recuperar quan hi haja una fallada d'especulació. Tot això fa que siga una estructura gran i complicada. Construir-ne una de nova (adaptant-la de la que ja es fa servir) seria una tasca que pot prendre prou temps, i per les limitacions que tenim de temps hem considerat que seria excessiu.

D'altra banda, també cal esmentar que construir una nova cua afegiria molta lògica redundant; a més a més tenir una cua discretament de flotants necessitaria una longitud pareguda (si més no, un poc més petita que la d'enters). Això faria que s'afegira una sèrie de lògica que ocuparia un volum considerable d'àrea.

Un últim aspecte a tenir en compte en aquesta decisió ha estat que una cua discretament de flotants no es faria servir per a res en un context exclusivament d'enters, i quedaria parada sense afegir cap funcionalitat i consumint energia.

Per tot això hem decidit ampliar la funcionalitat de la cua d'enters, perquè passe a ser una cua d'instruccions (*Instruction Queue*, IQ); per això s'hauran de modificar els vectors control, i que puga tenir suport per a instruccions *fused*, i que puga guardar informació d'escriptura/lectura sobre registres de flotant. Aquestes modificacions faran que haja un poc d'espai lliure a l'hora de guardar instruccions de coma flotant, que faran ús de vectors de control més petits, però tampoc és massa problema, ja que aquesta àrea s'hauria d'ocupar amb les extensions F i D o sense elles.

Per incloure l'FPU, *FPnew*, haurem de construir un nou component que s'encarregue de prendre els vectors de control i adaptar-los per que puga interpretar els senyals que contenen. També hem decidit crear una lògica que permeta les operacions de moviments de dades entre registres de manera separada a la FPU que comentem, per la manera en que està implementada necessitaria unes latències que sabem que es podien millorar.

La Lògica de Bypass també s'haurà de modificar per tal que les operacions *fused* puguen rebre 3 fonts, i construir una ruta per fer arribar les operacions a la FPU desde la IQ, i desde la FPU al banc de registres unificat.

El Mecanisme de Recuperació no patirà canvis importants, però ha de donar suport a les operacions *fused* i als registres de l'extensió F.

A més a més se li haurà de donar suport a les operacions de lectura de banderes de flotants que s'emmagatzemen al CSR, respecte a aquest punt no aprofundirem massa, ja que queda més enllà del que s'espera fer a aquest projecte, però tot i això considerem important esmentar-ho, ja que són uns canvis que afectaran a la implementació.

3.4 Notes sobre Propietat Intel·lectual

Un aspecte que ha condicionat de manera significativa l'elaboració d'aquest document han estat les limitacions de propietat intel·lectual i secret industrial. A títol particular, ens hauria agradat poder afegir codi RTL directament sobre el treball, de tal manera que es poguera seguir directament sobre el document, això, desgraciadament no serà possible ja que alguns fragments (o la totalitat) de les estructures podrien ser susceptibles de ser

patentats, o com a mínim, preferiblement mantinguts en secret per l'organització en la que es desenvolupa la nostra tasca.

I és que ha estat cridaner, venint d'un entorn de programació, i havent experimentat fins ara majoritàriament sistemes lliures o de codi obert què tant limita l'activitat professional haver de dependre de llicències de tercers; que pel que hem pogut observar no són una despesa per a res despreciable al desenvolupament de projectes hardware.

CAPÍTOL 4
Disseny de la Solució

En aquest capítol explorarem, en primer lloc quina ha estat la proposta de modificació des d'un punt de vista arquitectònic, és a dir, entendre quina serà a grans trets la ruta de dades que s'implementarà; quins seran els components pels quals passarà una instrucció i quin rol ocuparan cadascun d'ells en la implementació.

Seguirem fent una exploració més en profunditat els detalls que s'hauran d'implementar a cadascun dels components, i també definirem quins seran els vectors de control que ens serviran per governar l'arquitectura i establir un control del flux d'instruccions adequat.

Finalment, farem un recorregut de les tecnologies i ferramentes que hem fet servir per dur a terme aquesta implementació de manera ràpida i eficaç.

4.1 Arquitectura de la solució

Si fem un seguiment de la ruta de dades que traçaran les instruccions F i D dins del processador veurem que es poden moure per tres possibles camins:

- Ruta d'operacions aritmètiques per flotants
- Ruta d'accessos a memòria
- Ruta d'operacions de CSR

4.1.1. Ruta Unificada

Totes aquestes seguiran el mateix camí fins arribar al *Mapper*, on divergeixen.

Totes les instruccions provenen en primer lloc de memòria; ací assumirem que estan en la cache d'instruccions; poden no estar-ho, però per la nostra implementació això sols causa un retard en l'entrada d'instruccions, però això es una qüestió que està resolta en començar el nostre treball i queda més enllà del que es pretén cobrir a aquest treball.

La primera fase per la que passen les instruccions és la de *Fetch*, en la que es sol·liciten les instruccions a la cache d'instruccions i es presenten com a disponibles al processador per començar a realitzar operacions sobre la instrucció. En el nostre cas, la fase de *Fetch* consta de dues parts separades en cicles diferents: en la primera part es fa la sol·licitud de la instrucció a la que apunta el PC, (i de manera implícita la immediatament posterior). En el següent cicle, la cache d'instruccions les envia perquè estiguen disponibles per continuar el processament.

La segona fase per la que passen les instruccions és la de Decodificació. En aquesta fase es prenen les instruccions com està definit que s'han d'emmagatzemar en memòria en l'ISA i es transformen en senyals de control específics per al funcionament del processador de tal manera que siguen els que més li convenen a l'implementador a l'hora de fer-ne ús. En el nostre cas particular, el motor de decodificació consisteix en una memòria (BRAM) en la que s'utilitzen els valors binaris de les instruccions a decodificar com a adreces d'una petita memòria en la que s'especifiquen totes les instruccions possibles. Immediatament, la BRAM treu els valors dels vectors de control que convinguen per a cada instrucció. Finalment, aquestes dades s'envien al mecanisme de renombrat.

El mecanisme de renombrat s'encarrega d'assignar als registres físics del processador la tasca d'emmagatzemar les dades que haurien d'emmagatzemar els registres lògics. No es fa servir un mapejat directe d'un registre físic per cadascun de lògic perquè d'aquesta manera es presenten conflictes de WAR o de WAW, i això limitaria el rendiment del processador, especialment per sistemes amb pipelines profunds. En el seu lloc, per cada instrucció de li assigna un registre (o més si en calgueren) al que escriurà i aquest registre actuarà amb el nom del registre lògic que incorporava la instrucció de la que ve per instruccions posteriors. Això estableix entre les diverses instruccions una relació de generador-consumidor entre les instruccions, en la que les escriptures generen les dades que en posteriors instruccions s'han de llegir (consumir). En l'esquema de renombrat de registres és així com es representen les dependències vertaderes (RAW), que no es poden evitar.

La següent fase per la que passen les instruccions és la de *Mapping/Dispatch* en la que s'empaqueta la informació de control necessària i s'envia a la cua corresponent. Per aquesta fase es fan servir vectors de control generats a la fase de decodificació per determinar per quina ruta s'ha d'enviar de dades ha de seguir la instrucció, ja que és a partir d'aquesta fase que les rutes que seguixen les instruccions es separen i ja no es mouen pel mateix camí com fins ara. Cada ruta de dades tindrà la seua pròpia cua.

4.1.2. Ruta Aritmètica de Flotants

Les operacions de flotants que facen ús de la FPU, passaran per aquesta ruta. La primera parada a aquesta ruta serà la cua d'instruccions unificada. Ací les instruccions esperaran fins que els registres lògics dels que depenen estiguen disponibles. En aquesta estructura, el camí més òptim tardarà dos cicles en recórrer-se. El primer és el d'assignació d'un espai a la cua. Evidentment, si la cua queda plena, s'haurien de prendre mesures extraordinàries. En el nostre cas, el que caldria fer és ordenar un *stall* en primera instància sobre el *Mapper*, i si escau a la resta del front-end.

El segon cicle serà el que comença amb la notificació de la disponibilitat de l'última de les dades necessàries, el qual implica que es pot fer *Issue* a l'operació perquè es duga a terme l'operació en fase d'execució. Entre aquests dos cicles pot passar una quantitat indeterminada (però acotada sempre finita) de temps, i limitada a dos operacions per cicle.

Ací s'han acomodat els espais necessaris i les estructures lògiques necessàries per poder introduir operacions que depenen de fins a tres registres (enlloc dels dos com a màxim que teniem en implementacions prèvies), i poder acomodar les metadades que fan servir les estructures pensades per operacions de coma flotant, ja que prèviament sols es feia servir per operacions aritmètiques d'enters.

El següent pas és la lectura dels registres de dades desde el banc de registres. Ací s'ha afegit la possibilitat de llegir 3 registres per operació, enlloc de les dues com a màxim que necessita l'estàndard RISC-V d'enters.

És interessant notar un curtcircuit que ocorre i que afecta aquesta fase: Quan una instrucció arriba a WB, aquesta escriu al banc de registres, però en la implementació que hem descrit, la lectura de registres ocorreria dos cicles més tard, és a dir, s'afegiria una latència de dos cicles a totes les instruccions aritmètiques — de coma flotant o d'enters —. Per resoldre això, es notifica de manera prematura la disponibilitat de dades. L'execució de cada operació té una latència fixa. Així, si una operació executa en n cicles, la notificació de disponibilitat ocorre en el cicle $n - 2$. La nova operació arribarà mentre s'esta escrivint el resultat de l'operació antiga. En detectar-se aquesta coincidència se li entrega la dada directament sense llegir el registre que contindrà la dada.

Una vegada totes les dades estan disponibles, la fase d'execució pot començar. Tota la fase d'execució ocorre dins del motor d'execució. Ací dins trobem tot tipus d'estructures de càlcul: estan les FPUs que farem servir, però també les ALUs per les operacions d'enters, i la BU per les operacions de salt. El primer cicle de la fase d'execució per totes les instruccions és el d'assignació d'una unitat d'operació.

Una unitat d'operació és una estructura en la que s'inclou una ALU i una FPU. Es van decidir agrupar per tal d'evitar modificacions greus al motor d'execució i al bus de WB. Això té implicacions reals per al rendiment, desde el punt de vista d'implementació i és que una unitat d'operació sols podrà tenir un port de sortida i un d'entrada.

En el cas del camí de flotants, les dades s'adapten perquè puguen ser interpretades per la FPU correctament.

El criteri d'assignació és relativament senzill: s'intentaran assignar les operacions a la (nominalment) primera ALU, i si està plena, se li assignarà a la segona. Una vegada l'assignació s'ha dut a terme pot començar l'execució de l'operació pròpiament dita.

Aquesta separació entre la fase de lectura de registres i la d'assignació ha estat necessària per mantenir una freqüència de rellotge alta, ja que si bé pot semblar senzilla la lògica d'assignació és força complexa. I s'ha decidit mantenir separada de la lògica d'execució per evitar la generació de camins crítics que passen per una unitat d'execució.

Com s'ha comentat, per operacions de llarga latència, la notificació de disponibilitat de dades ocorre dos cicles abans de completar-se. Si una operació té un cicle de latència, aquesta notificació ocorrerà en el primer cicle de la fase d'execució.

La fase de WB comença a la sortida de cadascuna de les ALUs i FPUs. Les dades surten del motor d'execució i són emmagatzemades en el banc de registres, i marcades com a completades a la cua d'instruccions.

Finalment, quan són al capçal de la cua d'instruccions, i són les instruccions més velles del ROB, se'ls aplica la fase de *Commit* i són alliberades les seues entrades al ROB, la cua d'instruccions i els registres dels que llig, si cap instrucció posterior depèn d'ells.

4.1.3. Ruta de Memòria per a Flotants

De la mateixa manera que les operacions aritmètiques, les operacions de memòria, en sortir del *Mapper* passaran per una cua que permetrà la seua execució fora d'ordre. En aquest cas és la cua LSQ (*Load/Store Queue*). Té una operació similar a la descrita per la ruta de dades aritmètica, amb l'excepció que les operacions d'*store*, per poder-se executar se'ls ha d'haver fet *Commit*. Això implica haver rebut totes les dades necessàries, i haver resolt totes les especulacions que poden haver hagut anteriorment. Això sols es donarà quan siguen les operacions més antigues del ROB.

Les operacions de *load* poden continuar amb normalitat, però si depenen d'una direcció de memòria sobre la que operarà una *store* posterior es resoldrà localment fent-li un pas de les dades que l'*store* va a escriure en arribar a *Commit*.

Quan arriba el moment d'accedir a memòria —ja siga per *load* o per *store*— es farà un accés a la memòria cache L1. Les lectures d'aquesta cache tenen una latència de dos cicles, però és segmentada, pel que cada cicle es pot fer un accés a memòria nou. Les escriptures tenen una duració de 3 cicles.

Ací observem un curtcircuit paregut al que es veu amb Execució-*Issue* a la ruta aritmètica: en el moment en el que comença la lectura, es notifica de manera prematura la disponibilitat de les dades. Per la ruta d'execució això és una idea senzilla i pràctica per amagar latències, però a la ruta de memòria si bé permet incrementar rendiment, no és garantit que la dada que es sol·licita estiga a la cache de dades L1. Si no es dona el cas, s'haurà de recuperar l'estat anterior a la notificació de disponibilitat mitjançant el mecanisme de recuperació. Afortunadament, la latència és petita, pel que els canvis que pot ocasionar la notificació són limitats i tornar a l'estat anterior no serà massa costós.

En rebre la resposta de cache, s'escriu al banc de registres (sobre registres de flotants si cal), es torna a notificar la disponibilitat de dades (en aquest cas ja no és una informació especulativa) i es pot cometre l'operació.

4.1.4. Ruta per CSR/Barriers

Les operacions de CSR i les operacions de Barrier bloquen totes les operacions posteriors a elles. Això es fa de tal manera que quan arriba una operació de CSR (o de Barrier) al Mapper, aquesta és enviada al controlador de Barreres (*Barrier Sync Control a la figura* 3.2). Allí espera a ser la més vella del ROB. Mentre espera es fa que no entren noves operacions al front-end, i les que han entrat abans que arribara al controlador es descarten.

Per les operacions de Barrier, se'ls fa *Commit* i el PC s'apunta al $PC_{Barrier} + 4$, per les operacions de CSR, s'envien els senyals de control adequats. Això és important perquè l'extensió F de RISC-V inclou algunes operacions de CSR. Aquestes operacions o bé lligen el registre *fcsr* o bé el lligen i l'escriuen, això sols ocorrerà quan siga l'operació més vella del ROB. La lectura del CSR, escriptura i escriptura al banc de registres es fa tot en el mateix cicle, i al següent cicle la instrucció de CSR pot fer *Commit*, i alliberar la barrera.

Pel que fa a les banderes de Flotants, *fflags*, aquestes s'emmagatzemen temporalment a una entrada de control del ROB, i sols quan la seua instrucció associada arriba a *Commit* s'actualitza el registre de *fcsr* amb el valor corresponent, que és el seu valor previ o el valor de les banderes de la nova instrucció ($fcsr_{fflags}|HEAD_{fflags}$), per cadascuna de les banderes.

4.2 Implementació de la solució

En aquesta secció farem una exploració detallada dels canvis realitzats a cada component de les rutes de dades esmentades i les interaccions entre ells. En concret, farem èmfasi en la forma que s'han implementat els vectors de control que dirigeixen el flux de dades, i els detalls microarquitectònics que han dictat el com i com no de les descripcions RTL.

4.2.1. *Fetch* d'Instruccions

El mòdul de *Fetch* del processador Lagarto Ka es comunica amb la cache d'Instruccions per portar instruccions noves al core. En circumstàncies normals, es fa una sol·licitud i es reben 4 instruccions, les dues primeres s'envien a la fase de decodificació, i les altres es descarten. Si es reberen menys de dues instruccions, l'espai necessari per farcir les dues

vies que tenim cap a decode es farciria amb operacions nop. Acte seguit s'incrementa el PC en 8 unitats (ó 4 si sols es rep una instrucció).

Això seria el comportament típic però poden aparèixer esdeveniments durant l'execució en els que s'haja de modificar el PC: excepcions, interrupcions, recuperacions de context, salts condicionals o incondicionals, entre d'altres. En aquests casos, s'ignora la regla de canvi de PC establerta prèviament i es seguixen les indicacions de la unitat de control.

La interfície entre la cache d'instruccions i el motor de fetch es regeix per una FSM de dos estats: en el primer estat (*Idle*), la fetch es capaç de fer sol·licituds a la cache. El moment en el que fa una sol·licitud passarà al segon estat (*Wait*), en el que romandrà fins que reba resposta per part de la cache d'instruccions o el core notifique una *kill request*. La cache respondrà si té l'adreça sol·licitada, quan reba la línia que s'ha sol·licitat o en cas d'una excepció.

4.2.2. Decodificador d'Instruccions

El decodificador s'encarrega de traduir les instruccions RISC-V, d'acord a l'estàndard del manual a vectors de control capaços de ser interpretats pels mòduls del processador.

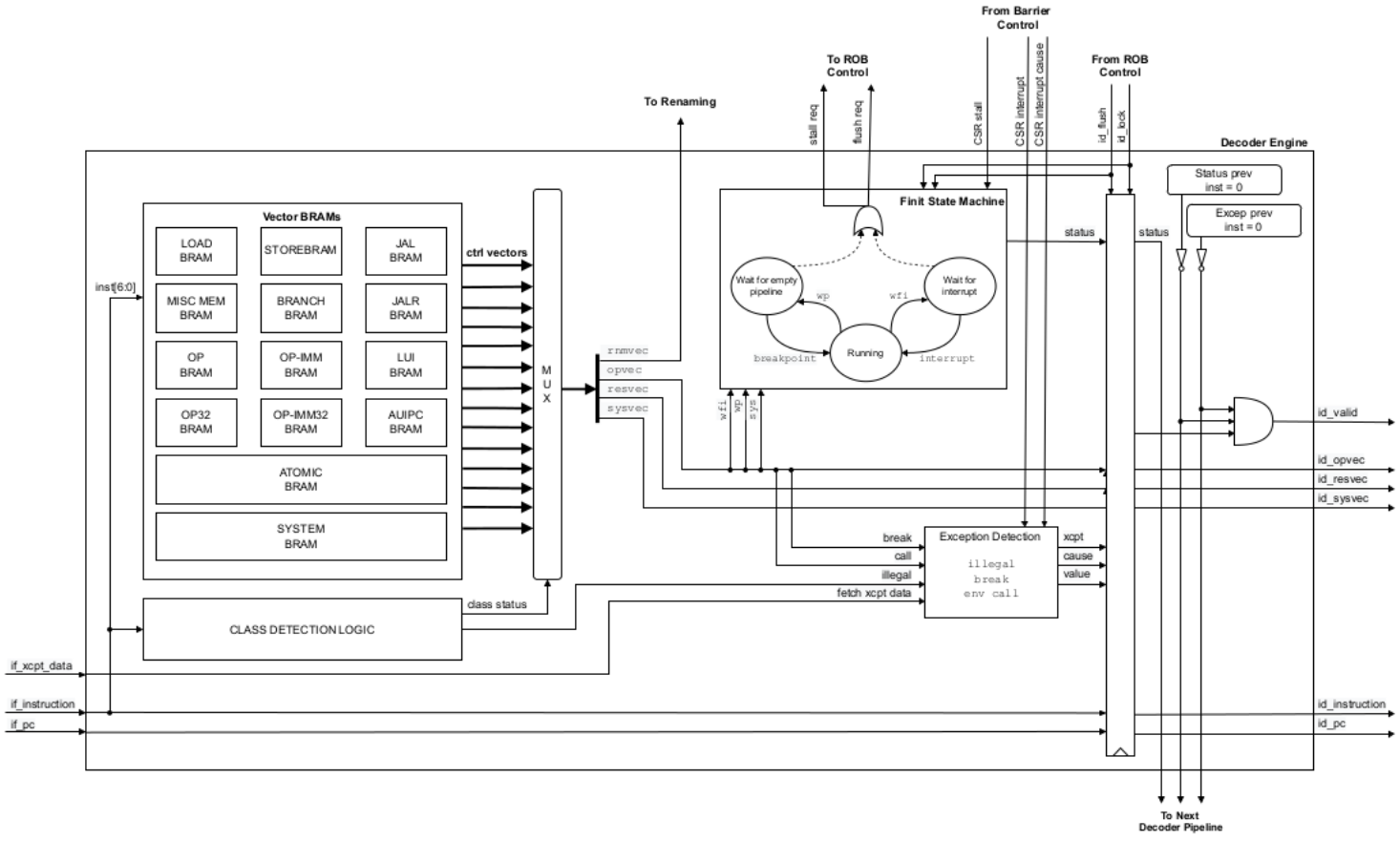


Figura 4.1: Ruta de dades del Decodificador de Lagarto Ka.[3]

Essencialment ací trobem una gran taula en la que segons el codi d'operació que porta la instrucció que ve de fetch, escollim una determinada una entrada. Aquesta entrada té associats els valors dels vectors de control `rnmvec`, `opvec`, `resvec` i `sysvec`. En paral·lel, el detector de classe indica la classe de la instrucció, per les extensions F i D tenim 3 classes distintes d'operacions: les artimètiques, les de memòria i les de CSR.

Posteriorment, es fa una detecció d'excepcions: això és si s'han rebut excepcions per part de fetch en el mateix cicle que s'han rebut les instruccions, o si les instruccions en processament han de generar una excepció (exemples d'instruccions que ho farien serien les instruccions ecall o ebreak), o si la instrucció introduida és il·legal, per exemple, si no està al conjunt d'instruccions que soporta el core. Si la primera instrucció genera una excepció, la segona serà descartada.

Vectors de Control

Els vectors de control definits que acompanyaran a les instruccions al llarg del control de flux del processador són: `opvec` (vector d'operacions), `rnmvec` (vector de renombrat), `resvec` (vector de recursos) i `sysvec` (vector de sistema).

El vector `opvec` és un vector de 17 bits dividit en dos subvectors. Els 9 LSB constitueixen el `cpuvec`, tot i que quan es supere el Mapper, ja que ja està en la seua ruta indicada ens referirem a ell com a `fpuvec`. En funció del context tenen un ús diferent, però per les operacions de coma flotant serveixen per incloure paràmetres de control que se li introduiran a la FPU. Cadascún té un significat diferent:

b_0 Fa ús de la FPU

b_1 Fa ús del camp rm de la instrucció

b_2 El destí té un format de doble precisió (1) / simple precisió (0)

b_3 La(es) font(s) tenen un format de doble precisió (1) / simple precisió (0)

b_4 Escriptura en registres de Flotants

b_5 Escriptura en registres d'Enters

b_6 Lectura de registres d'Enters (1) / de registres de Flotants (0)

b_7 Format d'enters INT64 (1) / INT32(0)

b_8 Fa ús d'enters

Els bits [11:9] es fan servir al mapper per identificar els codis d'operació i operacions de sistema especials. Aquests bits no es fan servir per instruccions d'enters, i queden sempre a 0; per altres tipus d'operacions tenen la següent funcionalitat:

b_9 Instrucció Fence

b_{10} Operació a executar

b_{11} Tipus d'instrucció

La resta de bits del vector [16:12] serveixen per indicar quina ruta de dades per la que s'ha de processar la instrucció:

b_{12} Ruta de dades d'enters

b_{13} Ruta de dades de memòria

b_{14} Controlador d'excepcions

b_{15} Ruta de dades de flotants

b_{16} Ruta de dades d'operacions vectorials*

*En procés de ser implementada

Respecte a aquest últim grup podem establir un clar paral·lelisme amb les classes d'instruccions que s'han esmentat: sols es faran servir els bits b_13, b_14 i b_15 per les instruccions d'F i D. En aquest cas sols una ruta estarà activa per cada cas, a excepció de les instruccions de memòria de les extensions F i D, en les que s'activen els bits b_15 i b_13, per

tal de facilitar el renombrat; el bit b_15 es desactivarà en la fase de renombrat per evitar problemes a futurs.

El vector `rnmvec` és un vector de 4 bits que indica quins registres de la instrucció s'han de renombrar de lògics a físics. Prèviament era de 3 bits, però les extensions F i D necessiten fer ús de 3 registres font, pel que s'ha extès a 4:

b_0 Activació renombrat registre destí

b_1 Activació renombrat registre font 1

b_2 Activació renombrat registre font 2

b_3 Activació renombrat registre font 3

El vector `resvec` és un vector de 8 bits que típicament serveix per indicar quines unitats funcionals farà servir una operació. En el cas de les instruccions F i D, el que puga necessitar queda limitat a la FPU, pel que s'utilitza per afegir paràmetres principals de control de la FPU:

$[b_0 - b_3]$ Codi d'operació per la FPU

b_4 Modificador de la operació

$[b_5 - b_6]$ Mode d'arredoniment

b_7 Operacions vectorials (en desús)

El vector `sysvec` és un vector de 6 bits pensat específicament per dur a terme les operacions de memòria. La seua funció principal és distingir entre operacions *Load* i *Store*, la resta de camps aporten informació auxiliar sobre com tractar la instrucció i com la durà a terme el mòdul de memòria. Per al cas de les extensions F i D sols es fan servir els dos primers bits i la resta queden a 0, no obstant, i per completitud s'ha decidit incloure'ls tots:

b_0 Operació Store

b_1 Operació Load

b_2 Operació Fence

b_3 Operació de Breakpoint

b_4 Operació de Sistema

b_5 Operació Atòmica

4.2.3. Renombrat de Registres

Al renombrat de registres quatre estructures diferenciades:

Register Alias Table

La primera és la RAT, una estructura que associa a cada registre físic un registre lògic. Diversos registres físics poden tenir assignat el mateix registre lògic, però a cada registre físic se li assigna un únic registre lògic. Pel que fa a la RAT, aquesta assignació és una memòria amb tantes entrades com registres lògics i en cada entrada s'emmagatzema una direcció física, la que s'ha escrit més recentment.

A la nostra implementació vam haver de discriminar entre direccions lògiques per registres de flotants i d'enters. Al binari de la instrucció, els registres d'enters i de flotants tenen les mateixes adreces, i la diferenciació entre ambdós resulta contextual, en funció de la instrucció de la que prové. Això fa que passar exclusivament la direcció d'un registre a

la RAT siga insuficient per diferenciar entre registres d'enters i de flotants, i acaba causant confusions entre registres d'enters i flotants.

Per resoldre açò vam afegir el bit b_{15} al principi de l'adreça lògica. Això no és una implementació que especifique RISC-V, però és una implementació compatible al que dicta l'estàndard, de la mateixa manera, es va duplicar l'espai de registres físics de 64 a 128, de tal manera que la diversitat d'adreces lògiques no suposara conflictes per falta d'adreces en contextos on enters i flotants conviuen.

Aquest canvi no queda limitat a la RAT, sinó que queda replicat a totes les estructures on s'adrece un registre físic, i al banc de registres, on s'ha hagut d'ampliar el nombre de registres.

La nostra RAT en concret, té 8 ports de lectura, i 2 d'escriptura. També té la possibilitat de recuperar-se fent servir els salts com a checkpoints. Per als registres font s'ordena la lectura de l'adreça de la RAT amb el valor del registre lògic de la instrucció, i treu una direcció física. De manera similar, per als registres destí, es pren la primera entrada de la FRL i s'escriu sobre la direcció amb el valor de l'adreça del registre lògic destí.

Així és l'operació típica d'una RAT per un processador no superescalar, però en processadors superescalars cal tenir en compte també dependències entre instruccions que entren a renombrat al mateix cicle, és per això que incloem les dues següents estructures:

La primera és la detecció de dependències vertaderes: ací es comprova si una instrucció A prèvia escriu un registre lògic que una instrucció B posterior llig. Això es fa comprovant que $Dest_a = Src_{bi}$, per tots els registres de B. Si hi ha cap dependència, la direcció física de la font que depén d'A es marca amb aquesta adreça física per B, en cas que no, es pren el valor que proporcione la RAT.

La segona és la detecció de d'escriptures *early-old*. Això és, quan dues instruccions que entren a renaming al mateix cicle tenen el mateix destí ($Dest_a = Dest_b$). Les dues instruccions llegiran de la RAT el mateix registre físic antic, això però no és el que dicta l'ordre de programa. Per resoldre-ho incloem aquesta estructura. Quan hi ha un conflicte *early-old*, a la primera instrucció se li assigna com a destí antic `Odest` el valor emmagatzemat a la RAT, i a la nova instrucció el valor que se li dona a com a nova destinació física, `Pdest`, a l'instrucció prèvia.

Aquestes tres estructures operen de manera concurrent seguint l'esquema mostrat a la Figura 4.2. La quarta estructura a tenir en compte és la Free Register List, en la que es guarden els registres que no estan en ús i, per tant, estan disponibles per ser usats per noves instruccions. Un registre sols entrarà ací quan la instrucció que l'ha sobreescrit (el tinga com a valor del camp Odest) haja fet commit. La FRL és una cua FIFO multiport, en el nostre cas en té 2 d'entrada i 2 de sortida, i amb tantes entrades com registres físics té el banc de registres.

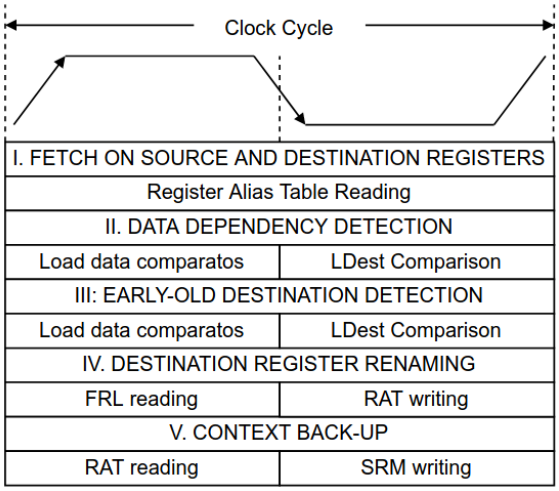


Figura 4.2: Esquema de temporització de la unitat de renombrat.[3]

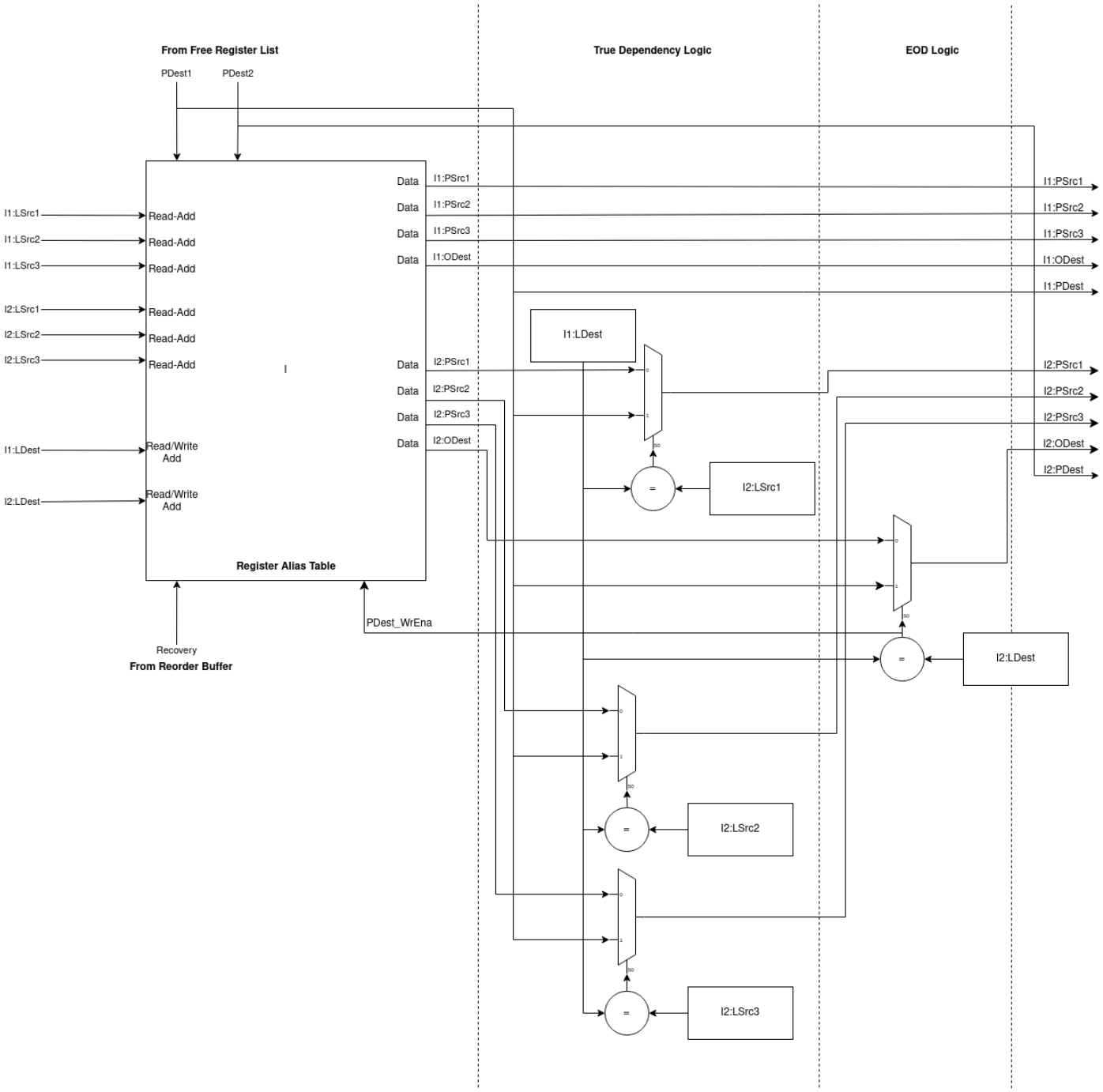


Figura 4.3: Diagrama de Blocs de les Estructures lògiques de Renombrat

4.2.4. Mapper/Dispatcher

La següent estructura i la última del frontend, que és en ordre és el Mapper. A aquesta estructura es preparen els paquets de vectors de control per enviar-los (o Despatxar-los) a les seues respectives cues. Els paquets que prepara se'ls anomena paraules. Les paraules es preparen a partir d'informació provinent de diversos llocs. Fins ara el que identificava cada instrucció i la sua antiguitat és el seu PC; a partir d'ara seran el `ROB_ENTRY`, un valor enter de 7 bits sense signe i un `BROB_ENTRY`, que identifica quin és el checkpoint de flux de programa del que depenen, en cas d'una fallada especulativa.

La primera tasca és identificar a quina ruta de dades s'haurà d'enviar una instrucció. Per això fem servir els bits [16:12] de l'`opvec` i identificar quins camps s'han d'omplir i

V	BrROB_Entry	ROB_Entry	PSrc1	PSrc2	PSrc3	PDest	LDest	Resvec	fpuvec
1	3	7	6	6	6	6	5	8	10

Figura 4.4: Estructura de les paraules per la cua de coma flotant

V	BrRob_Entry	Rob_Entry	Execode0	Execode1	cpuvec	resvec	ldest	psrc1	psrc2	pdest	imm
1	3	7	7	3	10	8	5	6	6	6	20

Figura 4.5: Estructura de les paraules per la cua d'enters

com. Per al cas de les instruccions d'F i D aritmètiques, es fa servir el bit b_1 per discernir si cal incorporar camp proporcionat per l'instrucció en els bits [14:12], o si fer servir directament el que es proporciona amb els bits [6:5] al `resvec`. quan el bit b_1 està encés, s'introduirà el valor que dicten els bits [14:12]. Noteu la diferència d'amplada en els camps: fent una ullada a l'estàndard podem comprovar que tots els modes d'arredoniment vàlid tenen el primer bit a 0, i l'únic que té el primer bit a 1 és el que indica que s'ha de sol·licitar el valor de CSR, pel que sols ens calen 2 bits per guardar el mode d'arredoniment a incloure. En les instàncies en les que s'haja de llegir el camp rm del registre `fcsr`, la lectura es produirà en un únic cicle, de tal manera que el Mapper sempre acabe l'operació en el mateix cicle.

Per al cas de les operacions FP d'accessos a memòria, aquests s'interpretaran com operacions LW, o LD (en funció de la longitud que tracten).

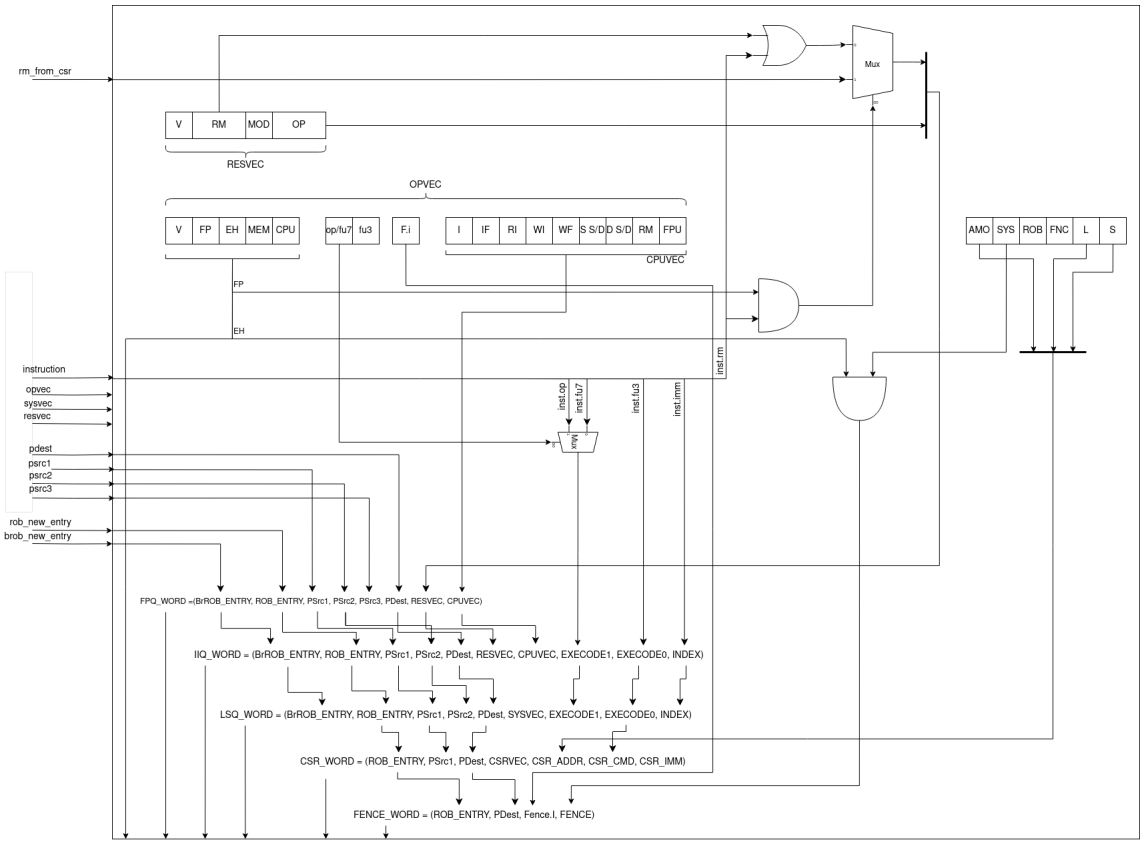


Figura 4.6: Diagrama de Blocs del mecanisme de Dispatch

La paraula que es genera per les operacions aritmètiques si que s'ha construit com a nova, podeu veure l'estructura a la figura 4.4. Aquesta paraula la interpretarà la cua d'instruccions (IQ) unificada. Per comparar, i sobretot per assegurar-nos que les dades

de control que necessitarem cabran a la IQ, haurem de comprovar que la longitud de les noves paraules no excedisca la longitud de les paraules que accepta (i emmagatzema) actualment la IQ. Els camps que farà servir són els que es mostren a la figura 4.5. Per diferenciar-los les paraules s'enviaran per interfícies distintes, una per les paraules d'FP i l'altra per les d'enters. Mai s'ocuparan tots els espais de les dues interfícies al mateix cicle. Això s'ha implementat així per oferir un suport senzill a la futura construcció d'una cua per coma flotant, però la implementació descrita actualment no la inclou.

4.2.5. Cua d'Instruccions

La Cua d'Instruccions (IQ) serà l'estructura que s'encarregarà de garantir un correcte funcionament de l'execució fora d'ordre. Podem distingir les següents tasques que haurà de realitzar:

- Emmagatzemar les dades de control per tal que s'executen les instruccions
- Garantir l'execució de les dependències en ordre
- Emetre les instruccions tan prompte com siga possible

D'aquesta manera, quan es rep una instrucció, aquesta s'emmagatzema a la primera entrada disponible. La instrucció esperarà fins que es satisfaguen totes les seues dependències. Es considerarà satisfeta la dependència quan es realitze la notificació d'escriptura prematura que es comentava al punt 3.2 sobre la implementació prèvia. Els canvis en aquesta estructura no són arquitectònics, ni massa profunds, no obstant han estat crucials per al funcionament adequat de les extensions.

Al tractar-se d'una estructura paramètrica, es va afegir suport per un registre font més (de 2 a 3), i es va incrementar en 1 bit l'adreça dels registres lògics (de 6 a 7 bits). Amb això la cua respectaria les dependències que necessiten les instruccions d'FP.

Aquesta cua prèviament era exclusivament per enters. Se li han afegit dos entrades i dos sortides per a operacions de FP, tot i això afegir entrades noves a la cua queda limitat a dos per cicle. Segons quina entrada seguisca la instrucció, s'emmagatzemarà un bit per marcar quina ruta ha de seguir al sortir.

La resta de bits necessaris per emmagatzemar les dades de control reutilitzaran l'espai que ocupen les instruccions d'enters. Afortunadament, l'espai necessari és menor que el que ocupen les operacions d'enters, pel que no caldrà afegir més espai.

El paquet d'informació d'enters que s'emmagatzema a la cua és el que es mostra a la figura 4.5. Aquest paquet té una longitud de 81 bits. En canvi, el paquet d'informació de flotants, que es pot veure a la figura 4.4 i té una longitud de 57 bits. Això ens ha permès no haver d'extendre les dades que s'han d'emmagatzemar, pel que no creix massa una de les estructures que més àrea ocupa.

L'assignació d'un espai de la cua tardarà un cicle, això suposant que hi ha espais disponibles. Sinó, les etapes del back-end hauran d'anar-se aturant fins que puguen entrar. Quan queden resoltes totes les dependències es permetrà que la instrucció faça *Issue*. Ara bé, en un mateix cicle diverses instruccions poden quedar amb les dependències resoltes al mateix cicle, quan es puga fer *Issue* de diverses instruccions es prioritzaran les instruccions més antigues.

Les instruccions sempre entraran a la cua en ordre de programa. Les instruccions sols s'esborraran quan facen *Commit*, pel que també s'esborraran en ordre. És a dir, que les instruccions dins la cua estan ordenades de més vella a més nova.

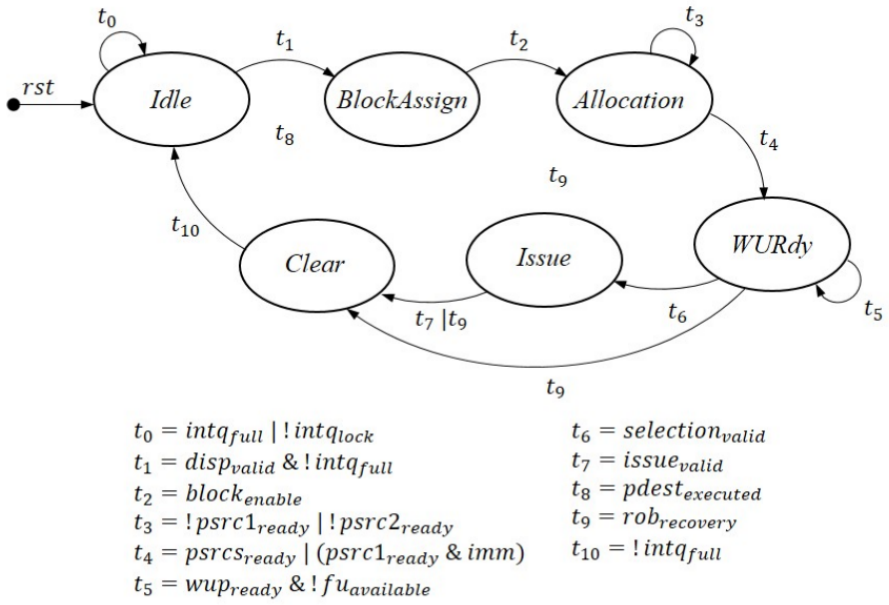


Figura 4.7: Diagrama d'estats pels que passa una instrucció a la cua d'enters.[3]

El diagrama d'estats pels que passa una instrucció és el que es mostra a la figura 4.7. Es passa pels estats *BlockAssign* i *Allocation* en un únic cicle. Per la comprovació inicial de si els registres dels que depen estan disponibles hi ha disponible un vector que conté quins registres estan disponibles per la lectura (han notificat la seua futura escriptura). Posteriors notificacions de lectura les rebrà directament del bus de WB. Quan satisfaga totes les seues dependències passarà a l'estat *WUIready*, fins que siga la instrucció més vella amb aquest estat. Aleshores se li farà *Issue* i podrà passar a execució.

Noteu com la notificació d'una escriptura ocorrerà en un cicle t, *Issue* es donarà lloc al cicle $t + 1$, i el primer cicle d'execució ocorrerà en $t + 2$.

4.2.6. Motor d'execució

Una vegada una instrucció arriba a execució ocorren dues coses simultàniament:

En primer lloc, es lligen els registres que s'hauran de fer servir. S'envien les direccions `psrc` que s'havien emmagatzemat a IQ. En el mateix cicle el Banc de Registres respon amb les dades pertinents i se'ls assigna una unitat funcional. Si les dades han de vindre per la ruta de curtcircuit en el seu lloc s'obtindran del bus de WB i el Banc de Registres no respondrà, ja que no té dades vàlides. Les operacions de FP s'accediran a un mòdul preliminar anomenat *FPU_Wrapper*. A aquest mòdul s'adapten les dades perque puguen ser llegides per la FPU, d'acord a les especificacions de l'Annex A. A aquest mòdul també es calculen les operacions de `fmv` (moviment de dades entre registres) per una ruta distinta que no passa per la FPU. Per diferenciar si haurà d'anar per la ruta de la FPU, o per la ruta per `fmv`, fem servir el bit b_0 del FPUVEC. Quan val 0 es tracta d'una operació de `fmv` i si val 1 s'haurà d'executar a la FPU.

Ruta per la FPU

Les operacions van acompanyades d'un bit de validesa. Quan es rep una operació vàlida, `FPUVEC.FPU` està encès, i no hi ha cap recuperació en procés, s'encendrà el senyal `in_valid_i` de la interfície de la FPU.

La FPU a més de tenir les dades de control que poden anar variant d'acord a la operació que es duga a terme, també té una sèrie de configuracions estàtiques que permeten adaptar el seu funcionament.

Aquesta FPU té suport per operacions de 16, 32, 64, i 128, però nosaltres sols fem ús de les operacions de 32 i 64 bits, i per enters té suport de 8, 16, 32 i 64, però nosaltres sols fem servir els enters de 32 i 64 bits. També permet desactivar el NaNBoxing, però l'hem deixat activat ja que l'extensió F de RISC-V el requereix. En principi permet fer operacions vectorials, però realment no hem considerat necessari suportar-lo fent servir aquesta FPU.

També permet segmentació. Nosaltres l'hem configurat perquè tarde 5 cicles per les operacions de suma i multiplicació (ADD/MULT), i per les de divisió i arrel quadrada(DIV/SQRT). Les de comparació, injecció de signe (NCOMP), i les de conversió (CONV) s'han configurat per tardar 3 cicles. És important comentar que el bloc d'operació per DIV/SQRT, permet estar segmentat, però les operacions no es duran a terme en paral·lel. En el seu lloc, la configuració genera una cua d'una determinada longitud que s'anirà executant en sèrie les operacions. L'element de DIV/SQRT tarda 23 cicles en retornar un resultat i fins que no acabe no començarà la següent operació. La resta d'element suporten la segmentació sense problemes.

A més a més val a dir que la FPU permet introduir dades sense que queden alterades durant la operació i viatgen per tota la FPU de manera sincronitzada amb la operació que acompanyen. Això està pensat per introduir un tag i es notifique al ROB quan acabe. La nostra implementació d'un ROB distribuït no ens permet fer servir aquesta implementació. En el seu lloc es fa servir per moure totes les dades de control en conjunt amb un tipus de dades dedicat.

En acabar la operació s'encén el senyal `out_valid_o`. Tan bon punt que s'encén, el resultat i les dades de control que l'acompanyen s'emmagatzemen en un registre interfase i s'encén el bit `out_ready_i` per tal que la FPU puga traure més resultats al següent cicle.

Això anirà directament a la fase WB. Pel que fa a les banderes d'excepció, quedaran emmagatzemades en un buffer per a banderes d'FP al ROB, a la IQ, i quan facen Commit se les enviarà al `fcsr`.

Ruta per fmv

Pel que fa a les operacions de fmv, s'ha afegit un registre que emmagatzema el valor de les fonts i les dades de control necessàries i a la sortida es fa el càlcul de NaNBoxing que requereixen les operacions fmv.

Per les fmv sobre 64 bits l'output és el mateix que el valor del registre font; per les operacions de 32 bits, l'output, que ha de ser de 64 bits es farceix amb 32 bits que valen 1, per les operacions que escriuen a registres de FP, i per als que no, es copia el bit b_{31} als 32 MSB.

Aquest és l'output que directament anirà a WB.

4.2.7. Àrbitre

Com heu pogut observar cada unitat funcional per flotants pot tenir diverses latències. En les nostres unitats funcionals sols pot entrar una operació per cicle. No obstant, es poden produir col·lisions quan dues operacions amb latències diferents han de sortir al mateix cicle.

És per això que hem introduit un element addicional de planificació dinàmica. Aquest element es troba just al principi del motor d'execució. Abans inclús de la lectura de registres.

Una manera de resoldre açò hauria estat no segmentar la unitat funcional, però això ens oferia un rendiment mínim, pel que vam optar per incloure aquest mòdul i fer servir la segmentació de la FPU — tot i que de manera limitada —. La idea general darrere de l'àrbitre són una sèrie de cues amb prioritat. De tal manera que són més prioritàries les instruccions amb latències més altes.

En la nostra implementació les instruccions viatgen d'esquerra a dreta, i han de viatjar com a mínim tants espais com cicles tarden en executar-se. Si anomenem 0 el cicle en el que una operació conclou execució, una operació de suma, per exemple, entrarà a l'espai 5, i en el mateix cicle se l'enviarà a executar. A aquest cicle li direm t. Si al cicle $t+2$ arriba una operació de comparació (3 cicles) entrarà a l'espai 3 de comparacions. Al cicle $t+2$, la suma ja haurà viatjat fins a l'espai 3. Si deixàrem ara que entrara la comparació a execució trobariem una col·lisió. El que fem és que la comparació espere un cicle i al següent ja entre a execució. Un exemple d'una traça el podem veure a la Taula 4.1.

Ací podem veure una sèrie d'operacions que entren en ordre segons el nombre que els acompanya. Les operacions 2 i 4 patiran una col·lisió de sortida, per això l'operació 4 haurà d'esperar fins que trobe un buit que omplir; de la mateixa manera, l'operació 6 haurà d'esperar fins al següent cicle per poder entrar al motor d'execució. Fins que l'operació 6 no puga ser llançada a execució, les operacions 7 i 8 tampoc podran. Una operació es llençarà a execució sols quan passe de la cel·la grisa de la seua fila a la següent. Una operació al entrar a la cua d'arbitratge anirà a la cel·la immediatament posterior a la grisa, si es pot llençar directament, o a la primera a la seua esquerra disponible.

De totes maneres, si les operacions 2 i 5 no existiren, les 4 i 6 no es podrien llençar simultàniament ja que, com hem dit, només hi ha una entrada. En aquest cas, es llençaria l'operació 6 per l'ordre de prioritat, i l'operació 4 haurà d'esperar.

En resum, existeixen tres tipus de conflictes que s'han de tenir en compte per saber com evolucionarà una taula d'arbitratge:

- Conflictes de sortida (operacions 2 i 4)
- Conflictes d'entrada (operacions 6 i 4)
- Conflictes de desplaçament (operacions 6 i 7)

El primer tipus és el més senzill d'avaluar: quan hi ha dos cel·les a la mateixa columna amb operacions vàlides, les dues arribarien al cicle de sortida simultàniament i això no està permès. El segon es detectarà quan hagen dues operacions a les cel·les de llançament, sols la menys prioritària es llançarà. El tercer es donarà quan la cel·la a la seua dreta està ocupada i no es desplaça.

La implementació d'aquest planificador s'ha fet amb una base de registres de desplaçament. Amb les implementacions esmentades, mentre sempre s'accepte la sortida al cicle en que arriba no hauria d'haver problema. Cal comentar que dalt o a la dreta de la línia que tracen les cel·les de llançament no hauran conflictes, i tampoc cal emmagatzemar totes les dades de control, ja que l'operació ja ha llançat i les dades ja estan viatjant per la ruta de dades. Tot i això si que cal guardar una marca de que està en execució i quant queda perquè acabe l'execució, per evitar conflictes com el que causarien les operacions 2 i 4 a la taula.

També podem fer servir aquest mòdul per fer la notificació prematura de WB que s'ha comentat en apartats previs. Observant si hi ha cap operació a la columna en que està

OP TYPE						
DIV/SQRT						
ADD/MULT			op5			op1
NCOMP	op8	op7	op6	op3		
CONV					op2	
FMV					op4	

Taula 4.1: Exemple d'una traça de com funciona l'àrbitre

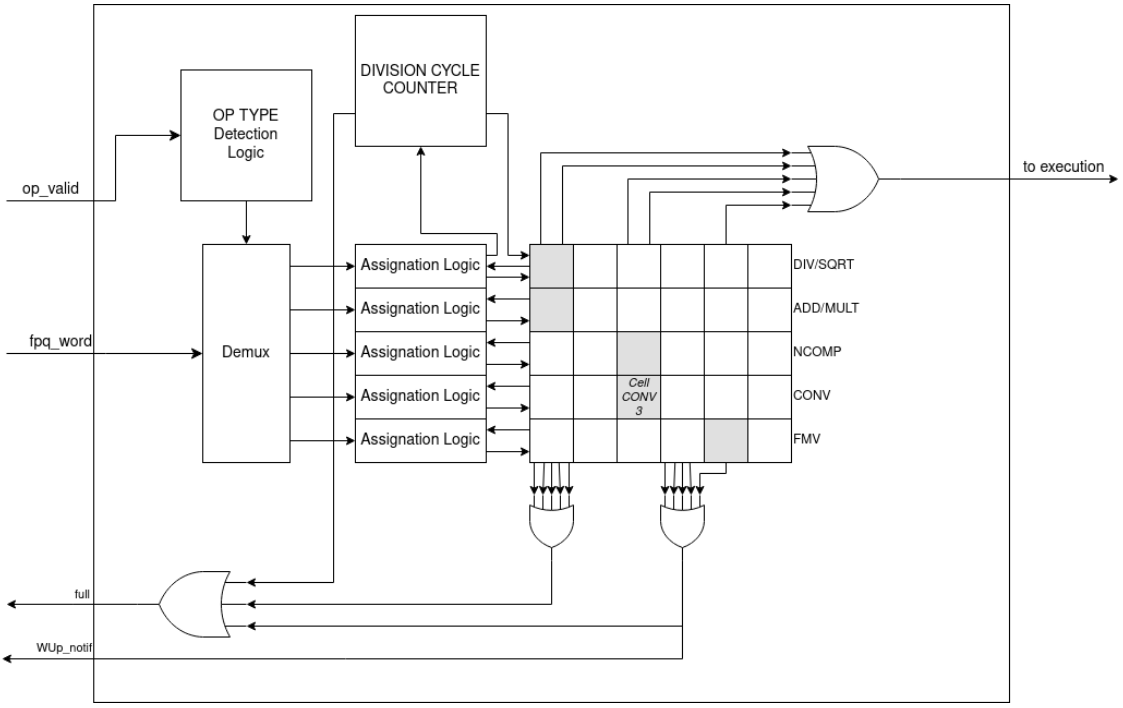


Figura 4.8: Esquema general de l'àrbitre de flotants

op3. Teniu en compte que en entrar una operació de `fmv` de latència 1 no é possible portar a terme aquesta tècnica. En aquest cas el que fem és fer la notificació en el cicle en que es llença l'operació. Això ens fa perdre un cicle que les dades queden parades al banc de registre per escriptures d'`fmv`.

Un altre punt a tenir en compte és el de les operacions de DIV/SQRT, que tarden 23 cicles. Per evitar afegir cel·les que rebran poca utilitat el que es farà quan arribe una operació DIV/SQRT és aturar l'entrada de l'arbitre i no permetre l'arribada de noves operacions a execució mentre l'operació visca. Les operacions que estiguen dins de l'àrbitre quan arribe una DIV/SQRT es permet que acaben, tot respectant les prioritats ja comentades. El moment en el que es llance a execució l'operació s'inicia un comptador de cicles, i es congela l'operació DIV/SQRT a la cel·la de llançament (sense que se la llance repetides vegades). Quan queden tants cicles com la següent classe d'operació per acabar l'operació es permet l'entrada de noves operacions i es descongela el desplaçament de la DIV/SQRT desdel punt que s'havia quedat.

Aquesta implementació s'ha decidit per poder emmascarar la latència que pren la DIV/SQRT. El bloqueig de la entrada s'ha inclòs per evitar l'arribada d'una DIV/SQRT mentre hi ha una altra en execució.

En arribar a les cel·les més a la dreta, les dades que puguen quedar dins de l'àrbitre es descarten. L'eliminació de les dades implica que una operació ha abandonat execució i està fent WB.

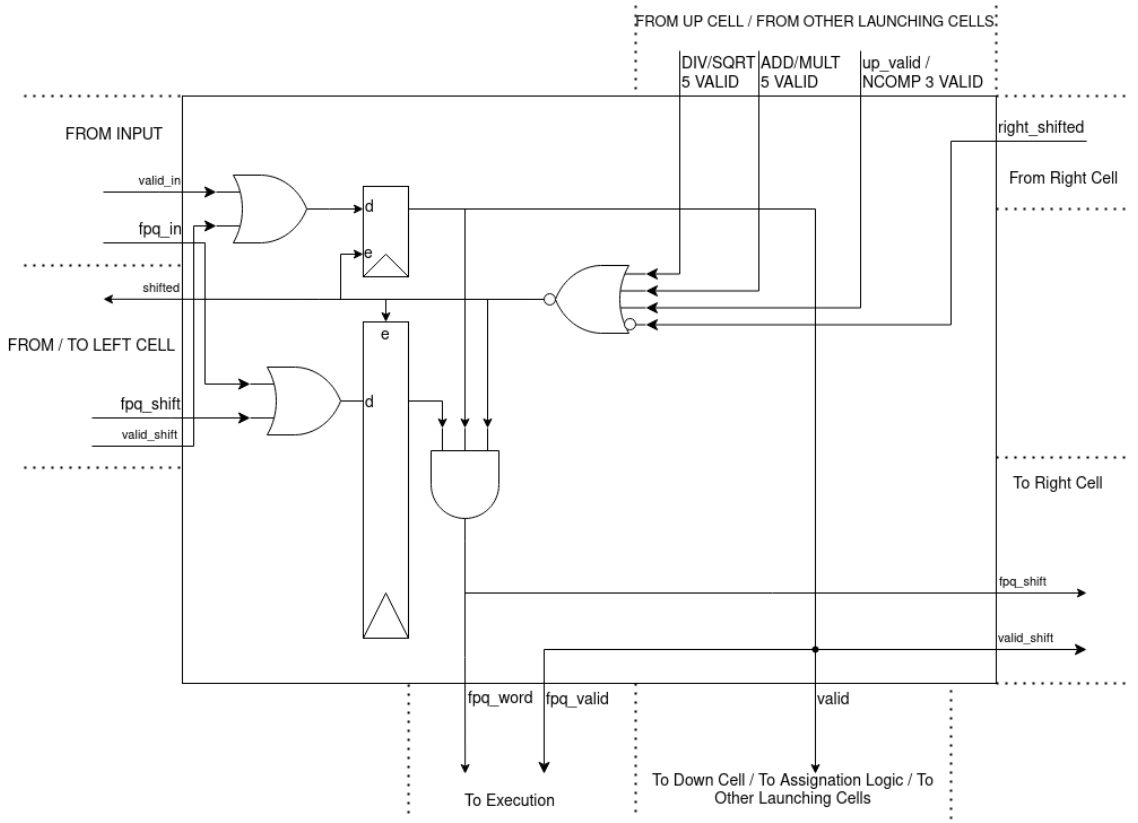


Figura 4.9: Esquema de la cel·la `CONV 3` de l'àrbitre

4.2.8. Unitat de Load Store

Pel que fa a les operacions de memòria de les extensions F i D, es mouran seguint la ruta de dades de memòria. En sortir del Mapper en ordre les instruccions de memòria s'envien a un buffer per operacions de memòria. De manera anàloga a com funciona la notificació de disponibilitat de registres per la cua d'enters, s'informa de les escriptures de registres de prematurament. Quan tots els registres font queden satisfets, s'envien les peticions de lectura a la cache de dades.

La cache de dades permet la lectura o escriptura d'una direcció de memòria per cicle. Per al cas de les lectures la resposta es donarà dos cicles després, si es disposa de la dada. I sinó, el cicle que hauria de respondre amb les dades es respondrà amb una notificació de fallada de cache.

La latència de dos cicles de resposta de memòria fa que es notifique prematurament la lectura prematurament el mateix cicle que s'envia la petició a la cache. Això es una notificació especulativa, és a dir, que no té garantit el seu correcte funcionament. La notificació haurà estat errònia si hi ha una fallada de cache. Quan es done una fallada de cache s'haurà d'activar el mecanisme de recuperació.

Pel que fa a les escriptures, queden al buffer de load/store fins que estan preparades per fer Commit. Això és, el seu `ROB_ENTRY` és al que apunta el `ROB_HEAD`, que és el marcador de la instrucció més antiga a la que se li ha fet Dispatch.

Les instruccions d'escriptura rebran una notificació de fallada si escau al segon cicle després d'enviar la seua petició a la cache. Al següent cicle podran efectuar Commit. En el cicle en el que s'efectua Commit, a la Cache es farà efectiva l'escriptura silenciosament. Mentre s'està efectuant una escriptura no s'accepten noves peticions d'escriptura ni de lectura, pel que no es fa ús del *pipelining*.

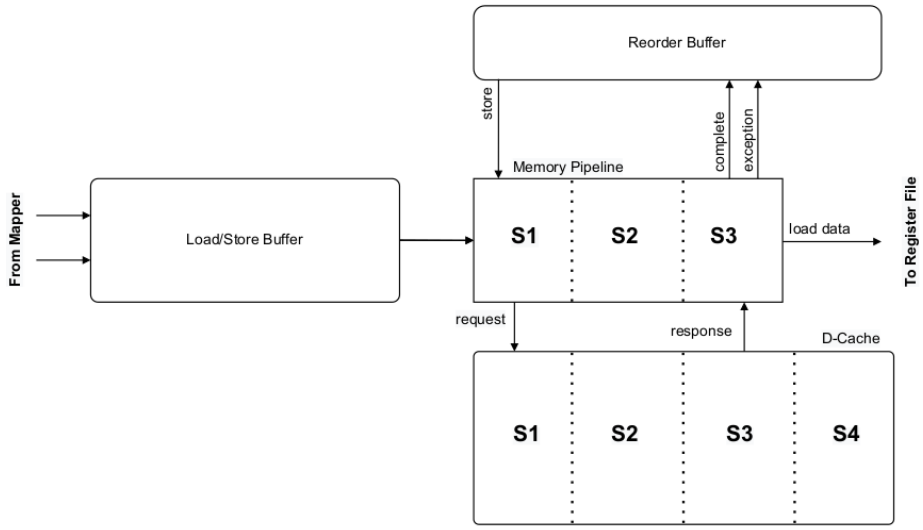


Figura 4.10: Esquema de Temporització de la unitat de load store [3]

Per facilitar la comprensió hem facilitat un esquema de temporització que es pot veure a la figura 4.10.

4.2.9. CSR i Excepcions

Els accessos als CSRs que té el processador es realitzen a través d'un buffer senzill entre el Mapper i l'interfície dels CSR que envia les instruccions quan estan preparades per fer Commit, és a dir, que com amb les operacions d'store les apunta el `ROB_HEAD`. Ací arribaran les excepcions detectades a Fetch i Decode com poden ser `ecall`, `ebreak`, `ifence`, o els accessos *misaligned* a memòria d'instruccions i les operacions d'accés a CSR. Mentre hi ha instruccions a aquest buffer, no es permet l'entrada de noves instruccions, i es congela el PC. Les operacions de CSR ordenaran el canvi del PC, bé per recuperar la instrucció que la seguiria segons l'ordre de programa, o bé per un salt que ordene la instrucció. Mentrestant, les instruccions que hi ha executant o a les cues d'instruccions segueixen sense alteracions (ja que són més velles) fins que fan Commit. Les instruccions que queden pel front-end després de l'entrada d'una instrucció al buffer seran descartades, ja que són més joves que la instrucció. Això fa que efectivament tota operació que accedisca a CSRs es tracte com una barrera.

Si una instrucció genera una excepció després de Dispatch, això és típicament durant execució o accés a memòria, es seguirà funcionant normalment i quan efectue Commit l'excepció generada passarà directament al CSR sense passar pel buffer previ. Quan es done aquesta situació tota instrucció que quede a qualsevol lloc del processador es descartarà, incloent instruccions que puguen quedar al buffer de CSR.

Tant les operacions de CSR com les excepcions són poc comunes, pel que maximitzar el seu rendiment no és una prioritat, però si que cal garantir un correcte funcionament ja que són les que garanteixen un correcte funcionament del Sistema Operatiu i altres mecanismes de control del programari.

A les extensions F i D les instruccions que es fan servir són per llegir o escriure el registre `fcsr`.

4.3 Ferramentes emprades

Pel que fa a la implementació, s'ha desenvolupat fent servir el HDL *SystemVerilog*. Açò és un llenguatge de descripció de hardware basat en *Verilog*, un altre HDL que al seu torn està inspirat pel clàssic llenguatge de programació C. *SystemVerilog* inclou la possibilitat de dissenyar, implementar, simular i testar sistemes electrònics. En el nostre cas ens hem centrat en la part de disseny RTL, en la que *SystemVerilog* és una ampliació de les funcionalitats de Verilog; això fa que *SystemVerilog* retrocompatible amb *Verilog*. Aquest llenguatge queda especificat a l'estàndard IEEE-1800-2009 [5].

Per facilitar el desenvolupament amb *SystemVerilog* hem fet servir l'entorn de desenvolupament de Visual Studio, desenvolupat per Microsoft, amb l'extensió per *SystemVerilog*. Aquesta instal·lació ens ha servit per simplificar l'escriptura de codi i evitar errors sintàctics que farien que el projecte no compil·lara.

Per comprovar que el disseny realitzat és correcte i s'adhereix a l'estàndard IEEE-1800-2009 hem fet servir el compil·lador de codi obert, Verilator que compil·la l'especificació en *SystemVerilog* a C per tal de generar executables vàlids. Quan el compil·lador dona errors, sabiem que l'espicificació no era correcta i s'havia de revisar. També ens ha servit per revisar bucles combinacionals que generaven senyals contínuament oscil·lants. Aquesta ferramenta també ens ha estat útil per la tasca de verificació.

Una vegada comprovat el correcte funcionament de la implementació, la nova versió actualitzada es pujava a un repositori privat en gitlab que serveix per mantenir un control de versions. Ací cada funcionalitat del projecte té una branca distinta de tal manera que es minimitzen les col·lisions. En acabar la verificació del sistema descrit a la branca, aquesta s'uneix a la branca principal, com a part preparada per la pròxima entrega del projecte.

4.4 Pla de Treball

En aquesta secció comentarem quin ha estat el pla que hem seguit per desenvolupar un disseny funcional amb les característiques descrites.

El primer pas va ser recopilar informació sobre quines eren les instruccions que s'anaven a implementar i les seues característiques. Això es va fer revisant amb deteniment la documentació de l'estàndard[2] i també com calia configurar la interfície de la FPU[4]. Això va ser un procés que va trigar una setmana i es va dur a terme desde el 12 de Desembre del 2022 al 16.

Una vegada recopilada la informació vam veure que la unitat de renombrat s'havia de revisar i parametritzar per poder incloure les funcionalitats de F i D. El procés de revisió de la Unitat de Renombrat per parametritzar-la es va allargar desde el 19 de Desembre del 2022 al 27 de Gener del 2023. Aquesta part del procés es va extendre en el temps per la falta de familiaritat amb el HDL que s'ha fet servir al llarg del projecte i ens va aprofitar com una primera presa de contacte amb ell.

Tot seguit vam passar a fer un disseny dels camps que havien de disposar els vectors de control i quins valors havien d'adoptar cadascún d'ells per cada instrucció de les extensions a les que anavem a afegir suport. El gros de la definició es va dur a terme el 30 de Gener del 2023 i el 10 de Febrer del mateix any, tot i que més endavant van haver revisions puntuals d'alguns valors. Aquests canvis es van incloure en el Decodificador simultàniament.

Ara el gros de la implementació va ocòrrer entre el 13 de Febrer del 2023 i el 31 de Març. En aquest termini podem distingir dos vies de paral·leles treball: en la primera es

va implementar el mòdul de *FPU_Wrapper* i tot seguit es va adaptar el mòdul d'execució per poder enviar instruccions a les FPUs. En la segona es va modificar el Mapper per donar suport a les instruccions de coma flotant, i tot seguit es va adaptar la cua d'enters per poder acceptar operacions sobre F i D, finalment també es va adaptar el mòdul de CSRs per incloure i donar suport a les operacions que ataquen el registre `fcsr`. També es van dur a terme modificacions menors en altres mòduls perquè tot quedara acomodat correctament

Amb això ja es tenia una versió preliminar de la ruta de dades i des del 3 d'Abril i fins al 28 es va fer un *debugging* profund de la ruta de dades seguint l'esquema que es comentarà al punt 5.1. La implementació fins aquest punt no semblava presentar fallades greus, però si que tenia un rendiment molt baix.

És per això que es va incloure el mòdul de l'àrbitre que va començar el seu desenvolupament el 2 de Maig i va continuar fins el 2 de Juny, tot seguit es va passar a verificar-lo i es va concloure una primera versió funcional de la ruta de dades amb l'àrbitre el 9 de Juny.

CAPÍTOL 5
Verificació i proves

5.1 La tasca de Verificació

La verificació en el desenvolupament de qualsevol aplicació és una part fonamental per oferir la millor experiència a l'usuari. Però pel que fa al disseny hardware és especialment important ja que un error no detectat quan s'envia el xip a fabricar-lo és extremadament costós de resoldre. És per això que durant la vida de projectes de desenvolupament hardware s'inclouen tècniques de verificació de manera extensiva, amb l'objectiu de capturar fallades tan prompte com siga possible, resoldre-les i estalviar en despeses que podrien haver comportat.

En el nostre cas la verificació ocorre bàsicament en 3 nivells:

1. A nivell de comissions.
2. A nivell de característiques.
3. A nivell *pre-release*.

La primera comprovació que es fa és avaluar la compil·lació de cada comissió al repositori fent servir Verilator o UML i executar una sèrie de tests bàsics que comproven l'execució correcta de les instruccions bàsiques de RISC-V [1]. Per al nostre cas, no es podia comprovar el correcte funcionament de les extensions F i D fins que no es va acabar de construir la ruta de dades, però si que es podia comprovar un funcionament adequat de les instruccions que es suportava en *releases* anteriors. Quan es detectava una fallada es s'observaven els arxius d'ones (veieu la figura 1.2).

Típicament una fallada a aquest nivell acaba ocasionant una aturada en el flux d'instruccions. En aquests casos el procés de *debugging* començava per buscar l'última instrucció correctament executada i l'última instrucció que entra al pipeline. Partint d'aquestes dues cotes recórrer la ruta de dades en busca de l'error que els últims canvis podien haver ocasionat. Una vegada es troba l'error es subsana i es torna a provar. Quan ja no es detecten errors es puja la comissió.

En pujar una nova comissió al repositori, el servidor executa comprovacions de linting amb les ferramentes de *Genus Synthesis*, Verilator i *Spyglass*. Tot seguit executa el joc de tests bàsic amb la compil·lació de Verilator. Aquest llançament automàtic de tests facilita el procés de verificació de cara al desenvolupador, se la anomena CI (*Continuous Integration*).

[1] Podeu trobar els tests bàsics al repositori https://github.com/riscv-software-src/riscv-tests

El segon nivell de comprovació ocorre quan una sèrie de característiques acaben la fase de disseny i implementació i es considera que estan preparades per unir-se al projecte general. En aquest cas es passen una sèrie de tests que poden ser representatius de càrregues de treball reals. En aquest projecte hem executat els tests bàsics i els benchmarks de riscv, tant els d'enters com els pensats per F/D. Quan un d'aquests tests fallava, revisàvem les ones, fent servir ModelSim o GTKWave i en busca d'errors que no s'havien contemplat. En aquest cas els errors solen resultar d'interaccions més complexes entre diversos sistemes del processador. En el moment que es detectava una fallada en l'execució, es corregia amb una comissió simple per errors que tenen solucions senzilles o creant una branca de solució de fallades per qüestions més complexes. En aquest nivell també hem executat alguns tests amb generació aleatòria d'instruccions. Aquests tests no estan pensats per buscar cap seqüència d'instruccions en concret que puga generar una fallada; sinó que generen instruccions de manera extensiva amb l'objectiu d'oferir tan bona cobertura com siga possible. Són molt llargs i es deixaven executant durant temps de poca càrrega als servidors del grup, típicament a la nit, és per això que se'ls anomena *nightly tests*.

El tercer nivell de comprovació ocorre una vegada s'han fusionat la branca amb la nova característica. Aquest nivell no queda contingut a un moment específic en el temps, en el seu lloc noves simulacions extensives s'executen periòdicament en busca de noves fallades. En afegir característiques posteriors també es faran comprovacions de les estructures prèviament incloses, pel que estructures més antigues queden més àmpliament cobertes que les més noves. Aquesta també és la fase en la que s'han pres les mesures de rendiment que informen a l'equip de quins sistemes cal retocar per tal d'incrementar-lo (o no). Les mètriques de rendiment acostumen a ser l'IPC i la freqüència, tot i que depenent del projecte se'n poden incloure d'altres.

El procés de generació d'aquests tests queda fora dels objectius que comentem a aquest document, però la literatura sobre el tema és àmplia.

Finalment, i com a prova extraordinària es va sintetitzar el disseny per provar-lo en una FPGA Alveo-U280 executant un arranc del sistema operatiu Linux sense cap incidència.

5.2 Rendiment i Resultats

Pel que fa al rendiment, hem fet servir els benchmarks que proporciona RISC-V per fer una avaluació, i tenir una estimació tant de l'IPC que és capaç d'assolir el sistema amb diverses càrregues. Adjuntem una taula amb els rendiments que ofereix tant la primera versió 5.1, que no incloïa l'àrbitre com la segona en la que si que s'inclou 5.2.

En aquesta taula podem apreciar un speedup global d'un 14%, per càregues d'FP amb l'addició de l'àrbitre. La natura de com opera la FPU no es permet incrementar de manera significativa el rendiment per les operacions DIV/SQRT, que és el que més limita el rendiment. Tot i això cal contemplar l'increment de rendiment per al test de *peak_flops*, en el que s'obté un speedup d'un 90%. Aquest test consisteix en la multiplicació i comparació de moltes operacions de manera consecutiva, pel que el test es pot beneficiar de manera important del *pipelining* de la FPU. Curiosament, el rendiment per als tests de *axpy*, *matmul_fp* i *spmv_fp* cau un 3%. Això es deu a que aquests tests fan un ús intensiu de les operacions de `fmv`. En la versió sense àrbitre aquestes operacions tenen un cicle de latència menys i això impacta el rendiment global.

Benchmark	Cicles	Instruccions	IPC
tests per enters			
fibonacci	15231692	10016971	0.66
histogram	17914	16526	0.92
matrix_mult	4100035	6785374	1.65
median	254581	207448	0.81
multiply	747040	617377	0.83
qsort	2650191	2130427	0.80
rsort	6952752	5542655	0.80
spmv	410853	293955	0.72
towers	277939	174054	0.63
vvadd	226282	230438	1.02
mitjana geomètrica			0.85
tests per coma flotant			
axpy	245629	143781	0.59
matmul_fp	1459210	942584	0.65
spmv_fp	138583	87262	0.63
peak_flops	70504	20042	0.28
peak_flops_div	252830	20030	0.08
peak_flops_mix	204772	25030	0.12
peak_mips	100951	200022	1.98
mitjana geomètrica			0.39

Taula 5.1: Resultats de rendiment sense l'àrbitre

Benchmark	Cicles	Instruccions	IPC
axpy	255840	143781	0.56
matmul_fp	1504716	942582	0.63
spmv_fp	142998	87262	0.61
peak_flops	36922	20043	0.54
peak_flops_div	217277	20029	0.09
peak_flops_mix	167461	25030	0.15
peak_mips	100951	200022	1.98
mitjana geomètrica			0.44

Taula 5.2: Resultats de rendiment amb l'àrbitre

CAPÍTOL 6
Conclusions

En aquest capítol farem un repàs dels objectius assolits, quines millores es podrien haver inclòs i quins problemes s'han trobat al llarg del desenvolupament del projecte.

6.1 Objectius Assolits

Els objectius que hem plantejat inicialment han quedat assolits amb diversos graus d'èxit:

Pel que fa als canvis arquitectònics per donar suport a les extensions RISC-V F i D podem afirmar sense dubte que han quedat completament assolits, ja que les simulacions i benchmarks ens confirmen que així és.

Pel que fa als objectius de verificació hem de dir que si bé tots els objectius han estat assolits de manera nominal, la verificació de la síntesi en FPGA amb l'arranc de Linux no és suficient, ja que aquesta execució fa un ús minso de la ruta de dades per Flotants, pel que caldria realitzar una exploració més en profunditat executant programes de referència més grans i complexos. Potser una manera de verificar el correcte funcionament seria executar els benchmarks d'EEMBC per FP compil·lats per RV64G en la FPGA.

Pel que fa a la inclusió de tècniques per millorar el rendiment, afegir l'àrbitre ha estat un pas positiu, però potser ha estat insuficient. Un canvi senzill que podria incrementar el rendiment seria tractar les operacions `fmv` com a operacions especials que no han de passar pel *FPU_Wrapper* i en el seu lloc queden resoltes com a operacions de moviment de dades un poc especials. Un altre canvi positiu hauria estat trobar una FPU que oferira *pipelining* per les operacions DIV/SQRT.

6.2 Dificultats que ens hem trobat

La dificultat més destacable que hem hagut de resoldre en el desenvolupament d'aquest projecte ha estat aprendre a fer ús de les ferramentes que es fan servir a la indústria.

Al grau l'exploració de entorns de desenvolupament per RTL és mínim i es tracta únicament en una assignatura. I tot i que la descripció del funcionament d'un computador modern al llarg de la carrera és adequada, l'aprenentatge de com es desenvolupen els components que l'integren és insuficient. Per suplir aquesta manca de coneixements pràctics, hem hagut d'emplear una bona part del temps del projecte en fer servir les ferramentes.

Sobre aquestes ferramentes val a dir que la documentació, per exemple, dels compil·ladors és escassa i, a la data de publicació d'aquest document, els més comuns no

implementen el complet de l'estàndard descrit per *SystemVerilog*, pel que ens hem trobat amb situacions en les que el compil·lador Verilator, per exemple, ens retornava errors de *no suportat*.

Pel que fa a la implementació, el problema més gros ha estat el seguiment de les instruccions durant la verificació. Llegir els arxius d'ones no és còmode des del punt de vista de comoditat humana i necessita d'un anàlisi en detall minuciós. No perquè conceptualment puga ser complex, que pot ser-ho en alguns casos, sinó perquè l'interfície estàndard que es presenta a la indústria no resulta massa *human-friendly*.

6.3 Futurs Treballs

De cara a futurs treballs, el processador Lagarto Ka passarà a tenir suport per l'extensió V (Vectorial), i a futurs la intenció es continuar i donar suport a l'extensió C d'operacions Comprimides. També ens hauria agradat perfilar amb més detall l'àrbitre per incloure el llançament d'operacions d'enters de tal manera que s'evitara fer ús de tècniques de *collision avoidance* que ens limiten el IPC.

També s'espera integrar el nucli Lagarto Ka com a part d'un multiprocessador. Per això primer caldrà desenvolupar els mecanismes de coherència de memòria adients per un SoC de les característiques que s'està desenvolupant.

Més a títol personal, aquest projecte ens ha servit per endinsar-nos en l'entorn del desenvolupament de hardware informàtic i el de arquitectura de computadors.

Bibliografia

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*. Agost del 2008.

[2] Editors Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. RISC-V Foundation. Maig del 2017.

[3] César A. Hernández Calderón. *SUPERSCALAR PROCESSOR DESIGN FOR EMBEDDED SYSTEMS*. Instituto Politécnico Nacional, Centro de Investigación en Computación, Doctorado en Ciencias de la Computación, Ciudad de México, México, 2021.

[4] Repositori de *FPnew - New Floating-Point Unit with Transprecision Capabilities*, OpenHW Group, ETH Zurich, 2023. El podeu veure a `http://americanhistory.si.edu/comphist/pr1.pdf`.

[5] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017*. Febrer del 2018.

[6] John L. Hennessy, David A. Patterson. *Computer Architecture: A quantitative approach (5th Edition)*. Waltham, Estats Units, 2012.

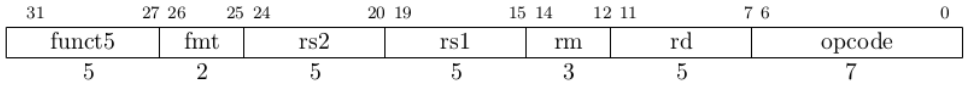
[7] Neil Weste, David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective (4th Edition)*. Boston: Addison Wesley, 2011.

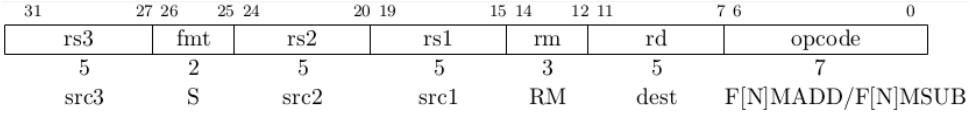
APÈNDIX A

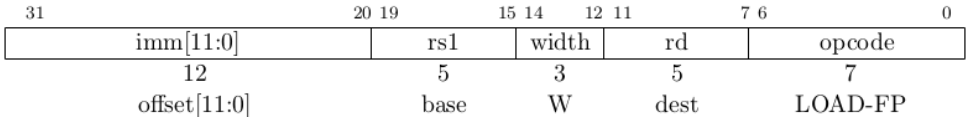
Interfície de FPU

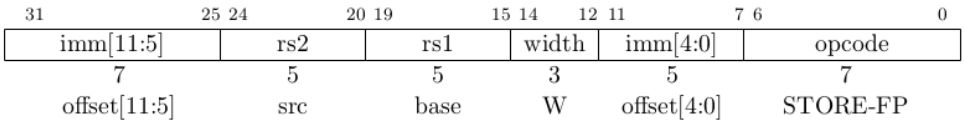
Especificació de la Interfície de la FPU

A.1 Senyals de la Interfície









<table>
<tr><th>Nom del Senyal</th><th>Tipus</th><th>Amplada</th><th>Direcció</th><th>Descripció</th></tr>
<tr><td>clk_i</td><td>control</td><td>1</td><td>SOC > FPU</td><td>Rellotge d'operacions</td></tr>
<tr><td>rst_ni</td><td>control</td><td>1</td><td>SOC > FPU</td><td>Bit de reinici dur. Actiu a nivell baix.</td></tr>
<tr><td>flush_i</td><td>control</td><td>1</td><td>ROB > FPU</td><td>Senyal de control de buidatge forçós desde el ROB</td></tr>
<tr><td colspan="5">Entrades</td></tr>
<tr><td>operands_i</td><td>dades</td><td>[0:2][W-1:0]</td><td>FPQ > FPU</td><td>Dades font</td></tr>
<tr><td>rnd_mode_i</td><td>control</td><td>3</td><td>FPQ > FPU</td><td>Arredoniments FP ($rm = inst[14:12]$), seguint l'estàndard de RISC-V</td></tr>
<tr><td>op_i</td><td>control</td><td>5</td><td>FPQ > FPU</td><td>Definició d'operació FP (camp <code>funct5</code>). Per F[N]MADD i F[N]MSUB, <code>funct7</code></td></tr>
<tr><td>op_mod_i</td><td>control</td><td>1</td><td>FPQ > FPU</td><td>Bit de modificació d'operació. Quan val 1, les operacions de suma, passen a ser restes</td></tr>
<tr><td>src_fmt_i</td><td>control</td><td>2</td><td>FPQ > FPU</td><td>Camp de format que codifica la precisió dels valors d'entrada de la operació:
<table>
<tr><th>Camp <code>fmt</code></th><th>Mnemònic</th><th>Significat</th></tr>
<tr><td>00</td><td>S</td><td>Precisió simple de 32 bits</td></tr>
<tr><td>01</td><td>D</td><td>Precisió doble de 64 bits</td></tr>
<tr><td>10</td><td>H</td><td>Precisió mitja de 16 bits</td></tr>
<tr><td>11</td><td>Q</td><td>Precisió quàdruple de 128 bits</td></tr>
</table></td></tr>
<tr><td>dst _fmt_i</td><td>control</td><td>2</td><td>FPQ > FPU</td><td>Camp de format que codifica la precisió dels valors de sortida de la operació anàlogament als de l'entrada</td></tr>
<tr><td>int_fmt_i</td><td>control</td><td>2</td><td>FPQ > FPU</td><td>Camp de format que codifica l'amplada dels enters que es fan servir en una operació amb enters:
<table>
<tr><th>Camp <code>fmt</code></th><th>Mnemònic</th><th>Significat</th></tr>
<tr><td>00</td><td>INT8</td><td>Enter de 8 bits</td></tr>
<tr><td>01</td><td>INT16</td><td>Enter de 16 bits</td></tr>
<tr><td>10</td><td>INT32</td><td>Enter de 32 bits</td></tr>
<tr><td>11</td><td>INT64</td><td>Enter de 64 bits</td></tr>
</table></td></tr>
<tr><td>vectorial_op_i</td><td>control</td><td>1</td><td>FPQ > FPU</td><td>Marca que una operació s'executarà entre vectors</td></tr>
<tr><td>tag_i</td><td>dades</td><td>Qualsevol</td><td>FPQ > FPU</td><td>Metadades útils pel control de flux de dades</td></tr>
<tr><td>in_valid_i</td><td>control</td><td>1</td><td>FPQ > FPU</td><td>Marca dades de l'entrada com a vàlides i disponibles per operar</td></tr>
<tr><td>out_ready_i</td><td>control</td><td>1</td><td>WBBUS > FPU</td><td>Marca que el component que ha de rebre les dades està disponible</td></tr>
</table>

Taula A.1: Senyals d'entrada de la FPU

Nom del Senyal	Tipus	Amplada	Direcció	Descripció
Sortides				
in_ready_o	control	1	FPU > RF / BPS	Marca la disponibilitat de la FPU a prendre la pròxima operació
status_o	data	5	FPU > RF / BPS	Marques d'excepció d'operació FP
result_o	data	[W-1:0]	FPU > RF / BPS	Resultat d'operació
tag_o	data	Qualsevol	FPU > RF / BPS / ROB	Sortida de metadades útils per al control de flux
out_valid_o	control	1	FPU > RF / BPS / ROB	Marca els valors de sortida com a vàlids perquè es prenguen per altres components
busy_o	control	1	FPU > FPQ	Indica que la FPU està realitzant una operació

Taula A.2: Senyals de sortida de la FPU

A.2 Notes de Cobertura

A.2.1. Protocol d'Entrada/Sortida a la FPU

Perquè una operació es prenga per part de la FPU tant `in_ready_o` com `in_valid_i` han de valdre 1. `in_valid_i` s'ha d'activar per part de l'usuari, i una vegada activat no s'ha de desactivar fins que el protocol complete l'acord. De la mateixa manera, perquè una operació quede completada, la sortida s'ha de llegir seguint el mateix protocol que per l'entrada: tant `out_valid_o` com `out_ready_i` han d'estar a 1.

Un acord completa quan ready i valid es lligen com a 1 durant un flanc de pujada de rellotge. El protocol és de *top-down*, pel que es permet que ready depenga de valid, però valid mai ha de dependre de ready.

A.2.2. Control

El vector de control principal és `op_i`, sobre el que es poden aplicar alguns modificadors:

- `op_mod_i` sols s'haurà d'activar per:
  - Restes, que substituiran a les sumes.
  - Conversions, per tal que el valor enter siga sense signe.
  - Operacions d'injecció de signe, per desactivar el NaN-boxing.
- `rnd_mode_i` s'interpretarà com a mode d'arredoniment per operacions aritmètiques i de conversió. Per les operacions d'injecció de signe, MINMAX i comparacions es farà servir com un modificador per alterar el tipus d'operació a realitzar.
- Per operacions aritmètiques `dst_fmt_i` i `src_fmt_i` haurien de valdre el mateix. Per operacions de conversió entre formats de flotant, tant `dst_fmt_i` com `src_fmt_i` es prendran com a valor vàlids, mentre que `int_fmt_i` s'ignorarà. Per operacions de conversió de flotants a enters i d'enters a flotants `src_fmt_i` o `dst_fmt_i` s'ignorarà, i quedarà reemplaçat pel camp `int_fmt_i`.

Noteu que `tag_i` es mou en conjunt amb les dades que referencia per dins del pipeline de la FPU sense patir cap alteració fins arribar a `tag_o`.

A.2.3. Excepcions

Les operacions de FP poden generar 5 tipus diferents d'excepcions, que quedaran marcats al vector de sortida en el següent ordre de MSB a LSB:

Mnemònic	Descripció
NV	Operació Invàlida
DZ	Divisió entre Zero
OF	Overflow
UF	Underflow
NX	No Exacte

Taula A.3: Possibles excepcions de FP

APÈNDIX B
Sobre ODS

Grau de relació del treball amb els Objectius de Desenvolupament Sostenible (ODS).

Objectius de Desenvolupament Sostenible	**Alt**	**Mitjà**	**baix**	**No procedeix**
ODS 1. **Fi de la pobresa.**			✓	
ODS 2. **Fam zero.**			✓	
ODS 3. **Salut i Benestar**		✓		
ODS 4. **Educació de qualitat.**		✓		
ODS 5. **Igualtat de gènere.**			✓	
ODS 6. **Aigua neta i sanejament.**		✓		
ODS 7. **Energia assequible i no contaminant.**		✓		
ODS 8. **Ocupació decent y creixement econòmic.**		✓		
ODS 9. **Indústria, innovació i infraestructures.**	✓			
ODS 10. **Reducció de les desigualtats.**			✓	
ODS 11. **Ciutats i comunitats sostenibles.**	✓			
ODS 12. **Producció i consum responsables.**	✓			
ODS 13. **Acció pel clima.**		✓		
ODS 14. **Vida submarina.**		✓		
ODS 15. **Vida d'ecosistemes terrestres.**		✓		
ODS 16. **Pau, justícia i institucions sòlides.**			✓	
ODS 17. **Aliances per aconseguir objectius.**	✓			

Taula B.1: Grau de Relació del TFG desenvolupat amb els ODS

Els ODS són una sèrie d'objectius de desenvolupament que s'han proposat desde les Nacions Unides que s'espera que d'assolir-se milloraran substancialment la vida de totes les persones del món, però que necessiten de la col·laboració de tantes institcions com siga possible.

En aquest sentit, el projecte esmentat a aquest document té una relació directa amb els objectius de desenvolupament d'indústria i infraestructura (ODS 9), per com s'espera que aquest projecte, com d'altres relacionats amb la indústria del disseny digital permeten el desenvolupament local de semiconductors tant a l'Estat Espanyol com a Europa. Com s'ha comentat al punt 1.1.3, les cadenes de subministrament que afecten als semiconductors suposen una infraestructura crítica per als estats moderns. L'ús del joc d'instruccions RISC-V no és purament econòmic, sinó que naix de dues vies compatibles amb els ODS. El primer és la reducció de consum energètic (ODS 11 & 12), fent ús d'un disseny amb extensions modulars es permet reudir el nombre de transistors de que fa un processador concret, ja que no ha d'implementar instruccions que no es faran servir en els entorns per als que es pensa, i per tant acabaria provocant la inclusió de transistors innecessaris.

L'altre punt al que afecta l'ús de RISC-V és el punt ODS 17 sobre la construcció d'aliances, i és que al tractar-se d'un estàndard obert, incentiva als seus usuaris i interessats a proposar canvis interessants. Aquests usuaris són d'arreu del món, pel que el disseny final ha de ser una d'acord i compromís. Aquest punt ODS 17 també afecta al nostre projecte concret, ja que el processador *Lagarto* es tracta d'una col·laboració entre el BSC i el IPN de Mèxic.

Comentar també que com està pensat per aplicacions d'HPC, es podria arribar a fer servir com a part d'un entorn de supercomputació per executar simulacions sanitàries, climàtiques, econòmiques o d'enginyeria. És per això que hem marcat que té una relació mitjana amb els ODS 3, 6, 7, 8, 13, 14 i 15.