Implementació d'un pipeline en un simulador de l'arquitectura RISC-V

Marc Orós Casañas

1/7/2019

Resum– Durant els últims anys, l'ús de sistemes de còmput de baix consum, com ara els telèfons mòbils, ha augmentat de manera significativa. El disseny de xips de baix consum té diverses dificultats, com ara el cost de la llicència a adquirir o la complexitat de dissenyar un xip des de zero. El projecte RISC-V té com a objectiu la definició d'una arquitectura oberta, amb usos com ara l'acadèmic, per a aprendre el funcionament d'un processador, o com a base pel disseny de processadors d'ús específic. Degut a l'actual manca d'alternatives pel que fa al hardware disponible per a desenvolupar software per l'arquitectura RISC-V, els simuladors són una gran opció per a poder escriure i testejar software mentre el hardware no està disponible a gran escala. En aquest treball s'implementa el model d'execució amb pipeline en un simulador de l'arquitectura RISC-V i es mesura l'augment de rendiment que proporciona l'ús d'aquest model d'execució.

Paraules clau– RISC-V, Conjunts d'instruccions, Simuladors, Codi obert, Pipeline, Hardware de baix consum

Abstract– In the recent years, the popularity of low power computing systems, such as smartphones, has skyrocketed. Low power silicon design has a lot of difficulties, such as the cost of licensing to modify a design, or the complexity of designing a chip from scratch. The RISC-V project has the goal of defining an open architecture, with uses such as academic use to learn the way a processor works at a low level, or to be the base design for application-specific processor designs. Due to the lack of options when it comes to available hardware to test and develop software for the RISC-V architecture, simulators can be a great alternative to be able to write and test software before the hardware becomes more widely and cheaply available. This project focuses on the implementation of the pipeline execution model on a simulator of the RISC-V architecture and measuring the improvement in performance that this execution model can provide.

Keywords– RISC-V, Instruction sets, Open Source, Pipeline, Low power hardware

1 Introducció

DURANT els últims anys hi ha hagut un gran augment en la utilització de hardware de baix consum, com poden ser els telèfons mòbils actuals o bé altres dispositius que utilitzen arquitectures ARM[1] o similars. Un dels problemes de les arquitectures com ara ARM o x86[2] (que són les arquitectures que s'utilitzen per als teléfons mòbils i ordinadors actuals) és que o bé cal pagar una llicència per a poder modificar l'arquitectura o bé directament no hi ha la possibilitat de modificar-la en absolut, sinó que l'única possibilitat és adquirir xips directament del fabricant. Això dificulta molt la possibilitat de poder dissenyar un xip per a utilitzar en una aplicació específica, ja que o bé hi ha un gran cost si es vol adquirir una llicència per a modificar un disseny ja existent, o bé s'ha de dissenyar el xip en qüestió des de zero, amb l'enorme esforç i complexitat que comporta el disseny d'un processador. L'arquitectura RISC-V[3] es crea per a intentar solucionar aquesta problemàtica.

1.1 RISC-V

RISC-V és un ISA[4] (Instruction Set Architecture) de codi obert basat en els principis de disseny RISC[5]. El pro-

• E-mail de contacte: marc.oros@e-campus.uab.cat
• Mención realitzada: Enginyeria de Computadors
• Treball tutoritzat per: Màrius Montón Macián (Microelectrònica i Sistemes Electrònics)
• Curs 2018/19

jecte va començar el 2010 a UC Berkeley, tot i que una gran part de les persones involucrades no estan relacionades amb la universitat. Aquest ISA es va dissenyar per a ús acadèmic, i es va escollir en comptes d'adoptar una arquitectura ja existent perquè havia de ser una arquitectura oberta i perquè altres arquitectures com ara ARM, OpenRISC, SPARC o MIPS tenien certes deficiències, explicades a l'article "Design of the RISC-V Instruction Set Architecture"[6].

Al no ser dissenyat amb una microarquitectura específica, RISC-V té el benefici que és molt versàtil. L'ISA RISC-V es pot implementar amb amples de paraula de 32, 64 o fins i tot 128 bits. Addicionalment, l'arquitectura té un component base, però es pot augmentar la funcionalitat dels dissenys segons les necessitats de la implementació específica utilitzant les diverses extensions, com poden ser:

- Extensió M: Multiplicació i divisió entera
- Extensió A: Instruccions atòmiques
- Extensió F: Punt flotant de precisió simple
- Extensió D: Punt flotant de precisió doble
- Extensió C: Instruccions comprimides
- Etc...

Un altre benefici d'aquesta arquitectura és que utilitza el concepte de "congelació" de certes parts de l'arquitectura. Si una arquitectura base o una extensió està congelada, vol dir que el seu funcionament bàsic no canviarà, i com a conseqüència, software escrit per aquestes arquitectures serà compatible amb les versions futures. Això és un gran benefici ja que permet a empreses escriure software i tenir la confiança que el software seguirà funcionant en versions futures de l'arquitectura.

Finalment, els dissenys originals utilitzen una llicència BSD[7], que permet modificar els dissenys originals sense haver de publicar els canvis, facilitant que empreses puguin fer dissenys amb arquitectures orientades a aplicacions específiques i poder-los comercialitzar sense haver de publicar les especificacions del disseny en qüestió.

Donat que RISC-V és una arquitectura molt recent, no és senzill adquirir hardware que utilitzi aquesta arquitectura, cosa que complica la creació de nou software per a la mateixa. Aquí és on entren els diferents tipus de simuladors, com per exemple els simuladors software, com el que s'utilitza com a base per a aquest treball, o bé implementacions hardware utilitzant FPGAs. Aquestes eines fan possible de forma senzilla poder implementar i testejar software per la arquitectura RISC-V mentre el hardware no és fàcilment accessible.

Aquest treball té com a objectiu observar a alt nivell com funciona l'arquitectura RISC-V i implementar un pipeline per a augmentar el rendiment del simulador i per a comprovar com diferents tipus de programes afecten el rendiment d'un processador que utilitza un pipeline.

2 OBJECTIUS

Els objectius d'aquest treball són, per una banda, comprendre el funcionament d'un simulador d'un processador per a poder expandir-ne la seva funcionalitat, i per una altra banda, implementar el model d'execució de pipeline[8] dins del simulador i mesurar la diferència de rendiment en comparació al model original.

L'objectiu a gran escala del treball és el de contribuir en un projecte de codi obert i proporcionar una implementació de la simulació una mica més propera al funcionament dels processadors actuals.

3 ESTAT DE L'ART

Aquest treball parteix d'un simulador de l'arquitectura RISC-V ja implementat[9]. Aquest simulador és de codi obert i està escrit en C++ utilitzant la llibreria *SystemC*[10]. Actualment el simulador és compatible amb l'arquitectura base de 32 bits i és compatible amb les extensions I, M, A i C. Com es pot veure a la figura 1 aquest simulador proporciona un mòdul de memòria, amb el qual és possible llegir i escriure dades, té un banc de 32 registres, un mòdul de *Trace*, que permet mostrar text per pantalla i els mòduls *Performance i Log*, que permeten obtenir certs paràmetres de rendiment i guardar dades rellevants en un arxiu de text sobre l'execució del simulador, com podrien ser errors, les instruccions que executa el simulador o l'estat dels registres afectats a cada moment.

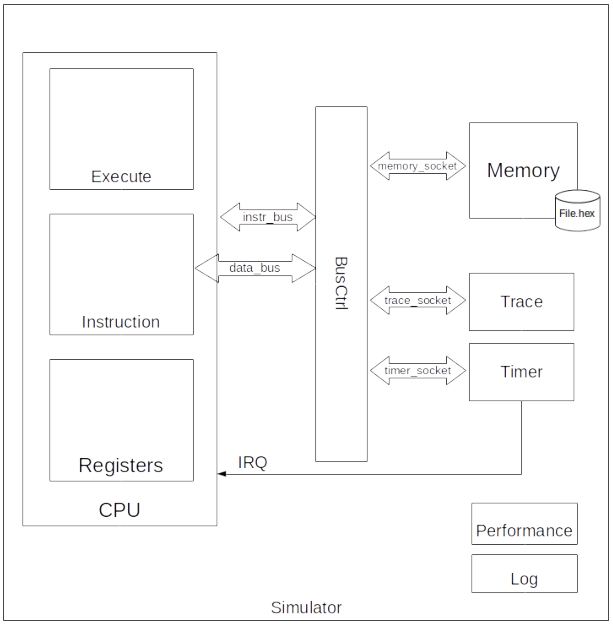

Fig. 1: Estructura del simulador

El simulador és capaç d'executar codi ensamblador, codi compilat en C (entre altres tipus de programes, és capaç d'executar el Sistema Operatiu en temps real FreeRTOS) i passa correctament la majoria de proves del conjunt *riscv-tests*[11].

Com s'ha mencionat abans, també existeixen simulacions a nivell de hardware de l'arquitectura, executant un disseny amb una FPGA. Els dissenys amb FPGA permeten executar i modificar de forma senzilla el disseny d'un processador en hardware real sense que sigui necessari fabricar els xips, amb tot l'esforç i cost que això comporta. També existeix un gran ventall de possibilitats pel que fa a simuladors software, amb implementacions tant de codi obert com amb llicències comercials, un exemple és Spike[12],

el simulador oficial de la fundació, altres exemples serien QEMU[13], Imperas[14] o FireSim[15].

3.1 Spike

Spike, que és el simulador oficial de l'ISA RISC-V, implementa un model funcional de un o més threads hardware RISC-V. Actualment Spike suporta les següents funcions de l'ISA RISC-V:

- ISA base RV32I i RV64I
- Extensió Zifencei
- Extensió Zicsr
- Extensions M, A, F, C, Q, C, V [16]
- Conformitat amb RVWMO i RVTSO (models de memòria)
- Modes de màquina, supervisor i usuari
- Capacitat de *Debugging*

De la mateixa forma que el conjunt d'instruccions i altres eines de RISC-V, el simulador està disponible de forma lliure i gratuïta a un repositori de GitHub[12]. Una funcionalitat interessant que té aquest simulador és la possibilitat de simular instruccions no incloses en el simulador base, implementant-les utilitzant plantilles ja existents. Això fa possible simular un processador de disseny personalitzat amb aquest programa sense la necessitat de dissenyar i fabricar un xip, o ni tan sols d'implementar-lo en una FPGA. Pel que fa a les possibilitats de *Debugging*, es pot utilitzar o bé gdb o el mòdul interactiu inclós amb el simulador.

Pel que fa a compatibilitat amb diferents tipus de programes, aquest simulador hauria de ser compatible amb pràcticament tot el software existent que utilitzi les extensions suportades pel simulador. De fet, actualment aquest simulador és capaç d'executar un sistema operatiu Linux complet.

4 METODOLOGIA

Per a completar aquest treball, s'ha dividit la feina en 5 etapes: la definició de la implementació, on es defineix quines etapes tindrà el pipeline i com funcionarà; la preparació prèvia, on es fan els preparatius per a tenir un entorn de programació adequat per a començar a treballar amb el simulador; la implementació del pipeline, on s'escriu el codi de cada fase; una fase de testeig, on es comprova que el funcionament del programa segueix sent correcte i coherent amb el model anterior, i finalment la fase de recollida de mètriques de rendiment, on es mesura el rendiment del simulador, es comprova com diferents tipus de programa afecten el rendiment i finalment es fa una comparativa amb el model inicial. Les següents seccions descriuen cada una d'aquestes fases.

4.1 Definició de la implementació

La primera tasca és definir les especificacions del pipeline que farem servir. L'augment de rendiment que proporciona un pipeline ve del fet que, al poder executar les diferents fases del pipeline en paral·lel, podem augmentar la freqüència de rellotge del processador (aquest augment està limitat per la duració de la fase més llarga del pipeline). Cal considerar que un pipeline comporta una petita penalització de rendiment quan es trenca el fil d'execució, com per exemple quan hi ha un salt. Per tant, com més etapes tingui el nostre pipeline, més rendiment podem extreure del mateix, però un major nombre d'etapes també implica un augment en la complexitat del funcionament del simulador.

Originalment el simulador fa tot el procés d'una instrucció de forma seqüencial, i sense cap mena de segmentació, cosa que fa que no sigui possible aprofitar el paral·lelisme de múltiples etapes. En aquest cas en particular s'ha decidit implementar un pipeline amb 3 fases, ja que és un bon balanç de rendiment i de complexitat. Les tres fases del pipeline tenen la funcionalitat següent:

- **Fetch:** Aquesta fase té tota la funcionalitat relacionada amb llegir la instrucció pertinent de la memòria de programa, descodificar-la i augmentar el valor del *Program Counter* (PC).
- **Execute:** Aquesta fase té tota la funcionalitat relacionada amb el còmput de les instruccions, accés a memòria i notificar al processador si la instrucció executada ha generat un salt.
- **Write-Back:** Aquesta fase escriu els valors calculats a la fase anterior als registres corresponents i fa tota la gestió d'escriptura i lectura de dades que encara no han sigut escrites al registre.

4.2 Preparació prèvia

El pas següent és la preparació de tots els components necessaris per a poder treballar amb el codi del simulador. Ja que treballarem amb programes que estan dissenyats per a ser executats a una arquitectura diferent a la de la màquina, hi ha alguna tasca addicional que no és necessària si tot el treball que fem és amb una única arquitectura. Les següents seccions defineixen els passos a seguir per a poder treballar amb el codi original i testejar el funcionament del simulador.

4.2.1 SystemC

Com s'ha mencionat anteriorment, el programa utilitza la llibreria SystemC per a dur a terme la simulació. Aquesta llibreria es pot adquirir de forma gratuïta a través de la pàgina del fabricant[10]. Un cop s'obté l'arxiu comprimit, s'ha de descomprimir. En comptes de ser un fitxer binari, aquesta llibreria proporciona directament el codi font, per tant, s'ha de compilar el contingut del fitxer comprimit, en aquest cas utilitzant l'eina *Make*[17] per a poder fer servir la llibreria. Cal destacar que per a simplificar les tasques de depuració s'ha compilat la llibreria amb la opció de depuració, però si l'únic objectiu fos executar el programa aquest pas no seria necessari. Un cop instal·lada la llibreria, cal

modificar la variable "SYSTEMC" del *Makefile* del simulador perquè el compilador sàpiga quina és la ruta on s'ha instal·lat la llibreria.

4.2.2 Entorn de programació C++

El segon pas de la preparació és triar un entorn de programació per a poder modificar el codi. En principi es podria utilitzar qualsevol sistema en el qual es puguin instal·lar els compiladors de C++ i la llibreria SystemC, i per a editar el codi es podria utilitzar qualsevol editor de text. En aquest cas en particular s'ha utilitzat el programa *Visual Studio Code*, ja que permet la depuració de programes escrits en C++ de forma senzilla i permet adaptar l'entorn de programació al projecte amb la utilització de scripts. Per tant tenim un únic programa que ens permet editar el codi, compilar-lo i depurar-lo de forma senzilla. Per a la depuració del programa s'utilitza l'eina gdb integrada dins del mòdul de depuració de *Visual Studio Code*.

4.2.3 *riscv-tools*

Ja que el simulador treballa amb codi de l'arquitectura RISC-V, ens és necessari tindre eines que ens permetin generar arxius executables per aquesta arquitectura a partir de codi font. Aquesta és part de la funció del conjunt d'eines *riscv-tools*[18]. Aquest conté una gran varietat d'eines software relacionades amb l'arquitectura RISC-V, com ara el simulador *Spike* mencionat anteriorment, una eina que enumera els codis d'operació que el simulador pot executar i les dues parts que utilitzem per aquest treball. La primera és *riscv-tests*. Aquest paquet conté múltiples tests que ens poden servir per a comprovar el funcionament del simulador, i van des de tests d'instruccions úniques fins a benchmarks sencers per a comprovar el rendiment. L'altra eina que utilitzarem és la *riscv-gnu-toolchain* que proporciona, entre altres coses, el compilador GCC per a generar codi RISC-V a partir de codi C i l'eina *objcopy*, que ens permet crear executables compatibles amb el simulador. Cal remarcar que actualment *riscv-gnu-toolchain* i *riscv-tools* són dos paquets separats, però formaven part del mateix conjunt d'eines quan es va començar a desenvolupar el treball.

4.2.4 Notes addicionals

El software anteriorment mencionat s'ha instalat dins del sistema operatiu Ubuntu 19.04. Per a poder accedir i modificar el codi del simulador i per a descarregar les *riscv-tools* s'ha utilitzat el sistema de control de versions Git[19]. La versió de la llibreria SystemC utilitzada és la versió 2.3.2.

4.3 Implementació del pipeline

En aquesta secció es definirà amb més profunditat com s'han implementat cadascuna de les fases del pipeline, la seva estructura, i quins canvis han sigut necessaris respecte al programa original.

4.3.1 Funcionament General

Per a cadascuna de les etapes del pipeline s'ha escrit una classe que conté la seva funcionalitat i dades necessàries. Cada classe té un mètode *run*, que conté el còmput que comporta cada etapa. El programa principal té una gran quantitat de mòduls, com es pot observar a la figura 1. Ens centrarem en els mòduls que s'han modificat en comparació a la implementació original. Dins del mòdul CPU, que és el mòdul principal on s'executen les instruccions del processador, hi ha un bucle on es crida cadascun dels mètodes *run* de cada etapa. Un cop s'han executat les tres etapes del pipeline, es crida un mètode *forward*, que prepara les dades generades per la pròxima iteració.

4.3.2 Etapa *Fetch*

El funcionament d'aquesta etapa és relativament similar a la part equivalent del codi original. La primera acció del mètode *run* de la classe *Fetch* és dur a terme una transacció, on es llegeix una instrucció de 4 Bytes de la memòria de programa. Un cop fet això, es comprova l'extensió de la instrucció i arriba el primer canvi. Originalment, es crida una funció *process_(extensió)_instruction()*, on es descodifica la instrucció i es fa tota l'execució corresponent a la instrucció. En el cas de la nova implementació, es crea una funció *decode_(extensió)_instruction()*, amb la qual guardem la instrucció descodificada i la seva extensió per a utilitzar-les a la pròxima etapa del pipeline. Dins del mètode *forward*, modifiquem el Program Counter en 2 o 4 Bytes en funció de l'extensió de la instrucció. Aquest augment es fa dins del mètode *forward* en comptes del mètode *run* perquè el funcionament sigui equivalent al simulador original, on s'augmenta el PC després d'executar la instrucció. La segona part que té aquesta etapa dins del mètode forward és transferir l'extensió de la instrucció decodificada i la instrucció en si als atributs corresponents de la classe *Execute*, perquè es puguin executar el cicle següent.

4.3.3 Etapa *Execute*

Aquesta segona etapa té més canvis que l'anterior. En primer lloc, s'han hagut d'afegir els atributs corresponents a l'etapa anterior a la definició de la classe. El primer atribut és l'extensió de la instrucció a executar, i els altres quatre atributs són un atribut per cada tipus d'extensió, que emmagatzemarà la pròxima instrucció a executar. S'utilitzen quatre atributs diferents en comptes d'un ja que en el codi original el tipus de dada d'instrucció és diferent per cadascuna de les extensions de l'arquitectura. En segon lloc, s'ha mogut el mètode *process_(extensió)_instruction()* de la classe CPU a aquesta classe (*Execute*). També s'ha modificat el comportament d'aquesta funció. En comptes de descodificar la instrucció que rep i executar-la, es rep una instrucció ja descodificada el cicle anterior i s'executa. El mètode *run*, en funció de l'extensió que utilitza la instrucció que hem adquirit al cicle anterior, crida el mètode *process* corresponent. Cadascun d'aquests mètodes llegeix quina instrucció és la que hem rebut, i crida el mètode específic de la instrucció perquè s'executi. Cadascun d'aquests mètodes té el seu funcionament específic, però el funcionament bàsic és el següent:

1. Accés als camps necessaris de la instrucció (valors immediats, registres, adreces de memòria...)
2. Còmput de l'operació
3. Escriptura del resultat al registre corresponent

Aquesta funció també notifica si hi ha hagut un canvi en el Program Counter, cosa que ens permet fer les accions necessàries si passa, com s'explica a continuació.

Ja que estem utilitzant un pipeline, hi ha dues situacions que s'han de tenir en compte, i caldran canvis perquè el programa segueixi funcionant correctament. El primer inconvenient és que, si executem una instrucció de salt on es realitza el salt, la instrucció que hem recuperat en l'etapa Fetch en el cicle actual i que executarem al cicle següent no serà la correcta, i per tant hem de perdre un cicle per a poder recuperar la instrucció correcta i executar-la, com es pot observar a la figura 2. Aquest comportament s'ha implementat afegint un bit a la classe que, quan es crida una instrucció que canvia el Program Counter, com podria ser un salt, s'activa, i quan aquest bit té el valor 1, l'etapa execute no executa cap instrucció, simplement torna a canviar el bit a 0. D'aquesta manera aconseguim que el simulador executi les instruccions en l'ordre correcte en cas que hi hagi un salt.

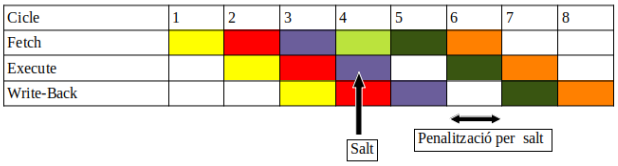

Fig. 2: "Pipeline flush"quan hi ha un salt en l'execució del programa

El segon inconvenient és que, originalment, quan llegim el Program Counter a l'executar una instrucció, el seu valor es troba a la instrucció que estem executant actualment. En el cas que utilitzem un pipeline, això deixa de ser cert, ja que el Program Counter es troba una instrucció més endavant. Això és un problema, ja que els càlculs originals que involucren el PC ara són incorrectes, amb el problema afegit que existeixen instruccions de 2 i 4 Bytes, i per tant no podem simplement restar 4 Bytes al resultat per a obtenir el resultat correcte. Per a corregir aquesta situació s'ha modificat el codi de cada una de les instruccions del simulador que fan càlculs utilitzant el PC, com podrien ser les instruccions de salt, i modificar els seus càlculs que utilitzen el PC, ajustant-los en 2 o 4 Bytes en funció de si la instrucció en si ocupa 2 o 4 Bytes. Un cop aplicats aquests canvis, el funcionament del simulador torna a ser correcte.

4.3.4 Etapa *Write-Back*

L'etapa final del pipeline s'encarrega de guardar els resultats generats en l'etapa *Execute* al seu registre corresponent. Per a implementar aquest sistema s'havien considerat dues possibilitats. La primera opció era afegir un atribut a la classe *Execute*, que representaria la sortida de la ALU, on es guardarien els resultats de la instrucció executada, i l'etapa de *Write-Back* s'encarregaria a cada cicle de guardar aquest valor al registre que toqui. El problema d'aquesta opció és que requereix modificar totes i cada una de les funcions originals perquè escriguin el seu resultat a aquesta "ALU" en comptes d'escriure directament al registre, i també cal implementar algun tipus de sistema per a saber a quin registre s'ha d'escriure el valor de la ALU. L'altra opció, que és funcionalment equivalent, i que s'ha utilitzat per a aquesta implementació és el sistema que es pot veure a la figura 3.

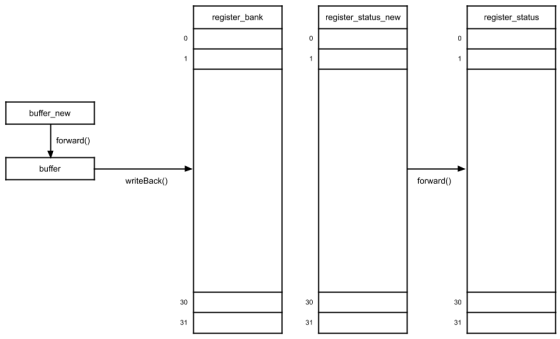

Fig. 3: Estructura del nou sistema de registres

S'ha modificat la classe *Register*, afegint-hi dos buffers i dos arrays d'estat. Aquestes estructures han d'estar duplicades perque volem saber l'estat del buffer i dels registres del cicle anterior. L'array d'estat està format per 32 elements que determinen quantes escriptures pendents queden a cadascun dels 32 registres del processador. Degut a aquest canvi i perquè el programa segueixi funcionant de forma correcta, ha calgut modificar les funcions d'escriptura i lectura de valors d'un registre.

La funció de lectura s'ha modificat, fent que, si hi ha una escriptura pendent al registre que llegim, llegim el valor del buffer, que seria el valor correcte, ja que el valor del registre estaria desactualitzat. En cas que no hi hagi cap escriptura pendent, es llegeix el valor del registre, que és el valor correcte.

Per l'altra banda, la funció d'escriptura funciona de la forma següent: el valor a escriure s'escriu al nou buffer per la següent iteració i s'actualitza l'array d'estat per a reflectir el fet que hi ha una escriptura pendent al registre que volem escriure.

Pel que fa a la funcionalitat d'aquesta etapa dins de la funció *forward*, hi ha dues parts. La primera, que seria l'etapa de Write-Back en si, comprova l'estat de cada un dels registres i si algun registre té alguna escriptura pendent, escriu el valor pendent al registre corresponent i actualitza l'array d'estat per a reflectir el fet que no hi han escriptures pendents. La segona part és passar el nou buffer al buffer actual i el nou array d'estat passa a ser l'array d'estat actual.

4.3.5 Canvis del programa externs a la implementació del pipeline

A part dels canvis mencionats als apartats anteriors, hi han hagut altres canvis no relacionats amb la implementació del pipeline, principalment afegits per a simplificar l'obtenció de mètriques de rendiment del simulador.

El primer canvi ha sigut modificar una part del mòdul de *Log* perquè no mostrés a l'arxiu de text el valor del Program Counter quan hi ha un "pipeline flush". El segon canvi ha sigut afegir un comptador de cicles al programa i un càlcul de l'IPC quan acaba l'execució del programa.

4.3.6 Canvi de freqüència de rellotge

Com s'ha mencionat anteriorment, un dels beneficis d'un pipeline és el fet que es pot augmentar la freqüència de rellotge, ja que estem limitats per la duració de l'etapa més

llarga, no la suma de la duració de les tres etapes. Originalment aquest simulador treballa amb un temps de cicle de 10 ns, equivalent a una freqüència de 100 Mhz. Per a la implementació amb pipeline s'ha escollit un temps de cicle de 4 ns, ja que si dividim el temps original de 10 ns entre 3 etapes obtenim 3,33 ns, i s'ha escollit 4 ja que, com que no sabem quina de les 3 etapes és la més llarga ni quant més llarga que les altres és, l'hem triat com a valor orientatiu proper a 3,33 ns, suposant que la variació de duració de cada etapa no és molt gran.

4.4 Testeig del funcionament del programa

El testeig del programa s'ha dut a terme en dues fases. La primera fase era el testeig del funcionament del programa mentre s'implementava una etapa del pipeline en particular. La segona fase era el testeig del funcionament del programa un cop el sistema sencer estava implementat, amb un ventall de programes de testeig més ampli i amb programes més complexos. Les dues fases s'expliquen amb més detall a continuació.

4.4.1 Testeig durant la implementació de les etapes

Aquesta fase del testeig es duia a terme al mateix temps que la implementació de cada fase del pipeline. Degut a la implementació del simulador, era possible implementar una etapa del pipeline i "connectar" el resultat a la resta del model original, i mantenir el funcionament del simulador. Per tant, s'han utilitzat dos programes senzills per a comprovar el funcionament correcte del simulador un cop implementada cada etapa. El primer programa és un programa senzill escrit en C que, utilitzant el mòdul *Trace* del simulador imprimeix els caràcters "He" per pantalla quan acaba l'execució del programa. Un cop fet un canvi al simulador, podem executar aquest petit programa i, tot i que no és prou complex per a dir-nos si el simulador funciona perfectament, és una forma senzilla de saber si alguna cosa no funciona bé. El segon programa, que es pot veure a la figura 4, està escrit en ensamblador i, tot i que està format per simplement 5 instruccions, és molt útil per a comprovar el funcionament de les dues últimes etapes, ja que hi ha instruccions que operen amb els registres. Per tant, si comprovem el funcionament dels registres amb el depurador, podem saber si l'etapa de *Write-Back* funciona correctament, i per l'altra banda, la instrucció de salt ens serveix per a saber si el nostre mecanisme de "pipeline flush" funciona correctament. Això es pot comprovar utilitzant el depurador i assegurant-nos que les instruccions que executa l'etapa *Execute* del processador es corresponen a les instruccions escrites al programa escrit en ensamblador.

4.4.2 Testeig de la implementació completa

Un cop hem comprovat que no hi ha cap error evident a cap de les etapes que hem implementat, s'han utilitzat 3 programes escrits en C per a comprovar el funcionament del simulador de forma més extensa.

4.4.2.1 Programa "prints"

Aquest primer programa de prova és un programa relativament senzill. Es defineix una variable temporal i es fan

```
.section .text
.globl _start
_start:
# Basic loop
  ADDI t1, zero, 10 # t1 to 10
  ADDI t2, zero, 1 #t2 to 1
loop:
  SUB t1, t1, t2
  BNE t1, zero, loop
  ECALL
# END
```

Fig. 4: Codi ensamblador del programa "BasicLoop"

múltiples operacions aritmètiques sobre aquesta. Els resultats de les operacions es mostren per pantalla utilitzant el mètode *print*, que utilitza el mòdul *Trace*. Aquest programa serveix per a comprovar el funcionament de certes operacions aritmètiques, el funcionament dels registres i dels salts. Addicionalment també utilitza la llibreria estàndard de C per la funció *sprintf*, fet que fa que l'executable resultant sigui molt més gran que els programes anteriors (300 Bytes del programa *Trace* contra 150 kB d'aquest programa). Executar aquest programa va servir per a detectar que a la instrucció "AUIPC" no s'hi havia fet la correcció del càlcul del valor del PC, i que només s'havia fet a les instruccions de salt.

4.4.2.2 Programa "benchmark"

Aquest programa és un banc de proves senzill escrit per a comprovar el funcionament del simulador, però també per a recollir resultats de rendiment. El funcionament d'aquest programa és el següent:

1. Multiplicació de dues matrius de mida 24x24 16 cops: permet comprovar que el mòdul de memòria funciona correctament, a part de les operacions de multiplicació i divisió

2. Bucle d'1 milió d'operacions NOP: permet comprovar el funcionament de les operacions de salt i ens serveix per a comprovar la penalització que comporten les instruccions de salt

3. Aplicar el procés de DCT i quantització (utilitzats en la compressió JPEG) a un bloc 8x8 aleatori 8 vegades: semblant a la primera part, comprova el funcionament de la memòria i s'executen operacions trigonomètriques i d'arrel quadrada. Degut a problemes amb el compilador (generava operacions de punt flotant, no compatibles amb aquest simulador) aquesta part del programa no s'ha pogut utilitzar

A part d'aquestes 3 parts s'escriuen per pantalla missatges sobre el progrés del programa i per tant es torna a utilitzar les funcions *sprintf* i *print*.

4.4.2.3 Benchmark "Dhrystone"

Dhrystone és un banc de proves sintètic amb el propòsit de ser representatiu del rendiment de processadors de propòsit general (CPU). El nom és un acudit relacionat amb l'algoritme de comprovació del rendiment anomenat Whetsto-

ne. Aquest programa utilitza crides a funció, indireccions a punter i assignacions, entre altres. A diferència del programa Whetstone, Dhrystone utilitza únicament operacions enteres, sense cap operació de coma flotant. El programa retorna un resultat de Dhrystones/segon.

Cal considerar que aquest programa no funciona completament amb el simulador, i retorna resultats incorrectes, però el simulador és capaç d'executar-lo de forma completa i, encara que no retorni resultats correctes, ens és útil per a comparar el rendiment amb la versió original del simulador

4.4.3 Resultats del testeig

Després de múltiples proves amb els programes descrits entre altres, s'ha arribat a la conclusió que el simulador segueix funcionant de forma correcta, o almenys de forma correcta fins al mateix nivell que el programa original.

4.5 Recollida de mètriques de rendiment

Un cop s'ha comprovat que el programa funciona correctament, s'han recollit mètriques de rendiment de les dues versions del simulador, utilitzant el mòdul *Performance* que proporciona el simulador. L'execució del simulador proporciona dos resultats, a més del text mostrat per pantalla del mòdul *Trace*, com es pot observar a la figura 5. El primer és un bolcat de l'estat final dels registres, mostrant els valors emmagatzemats a cada registre. El segon resultat són les mètriques de rendiment, que són les següents:

- Lectures a la memòria de programa
- Escriptures a la memòria de programa
- Escriptures a registre
- Lectures de valors de registre
- Instruccions executades
- Cicles completats
- IPC de l'execució del programa

```
**********************************
Registers dump
x0 (zero):           0 x1 (ra):          67076 x2 (sp):       67108703 x3 (gp):         123368
x4 (tp):             0 x5 (t0):          67524 x6 (t1):          4912 x7 (t2):              0
x8 (s0/fp):    67108863 x9 (s1):             0 x10 (a0):        116116 x11 (a1):       67104591
x12 (a2):      67108495 x13 (a3):       6855225 x14 (a4):        116116 x15 (a5):             0
x16 (a6):   -1610612736 x17 (a7):      67108323 x18 (s2):             0 x19 (s3):             0
x20 (s4):            0 x21 (s5):             0 x22 (s6):             0 x23 (s7):             0
x24 (s8):            0 x25 (s9):             0 x26 (s10):            0 x27 (s11):            0
x28 (t3):           -2 x29 (t4):             0 x30 (t5):             0 x31 (t6):             0
PC: 0x10608
**********************************
Simulation time 90877996 ns
# data memory reads (not implemented): 0
# data memory writes (not implemented): 0
# code memory reads: 22719500
# code memory writes: 0
# registers read: 36092514
# registers write: 17575046
# instructions executed: 21411314
# cycles completed: 22719500
Calculated IPC: 0.94242

Error: Execute: ECALL
In file: src/Execute.cpp:1320
In process: top.cpu.CPU_thread @ 90877996 ns
```

Fig. 5: Resultat de l'execució del programa amb l'executable "benchmark"

Addicionalment, el simulador també mostra si hi ha hagut algun error a l'execució. En el cas de la figura 5, es mostra un error en l'execució de la instrucció ECALL, però aquest és el funcionament normal ja que la instrucció "ECALL" és la que marca la finalització de l'execució d'un programa.

5 Resultats

En aquesta secció es mostren els resultats de rendiment de les dues versions del simulador. El codi de la versió modificada del simulador, incloent els tests que s'han utilitzat en aquest treball es pot trobar a GitHub[20]. Pel que fa a la metodologia, s'han executat 5 programes (Addition, Extended, benchmark, prints i Dhrystone) a les dues versions del simulador, l'original i la versió amb pipeline. S'han utilitzat les mètriques mencionades a l'apartat anterior per a comparar el rendiment de les dues versions del simulador.

5.1 Speedup de la simulació

En aquesta secció es compara el rendiment del processador simulat en la nova versió del simulador en comparació a la versió original

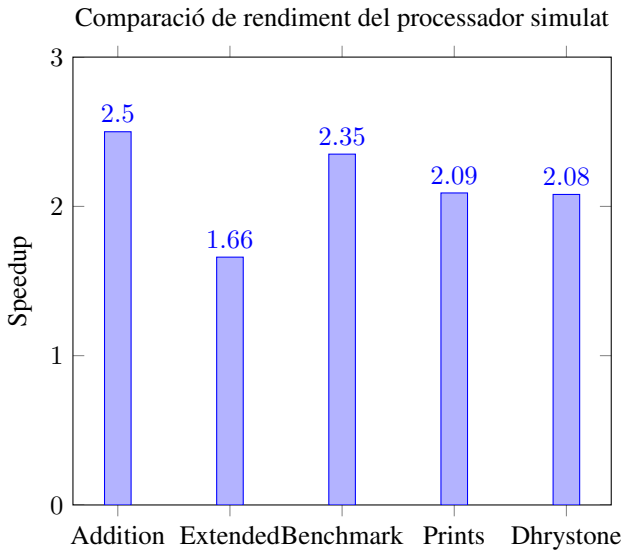

Fig. 6: Gràfic comparant la reducció de temps d'execució (speedup a nivell de simulació) entre la versió original del simulador i la versió amb execució amb pipeline

Com es pot observar a la figura 6, el rendiment millora, en promig, 2.13 vegades. Aquest resultat és bastant proper a l'speedup òptim de 2.5, que és el factor en què hem augmentat la freqüència de rellotge del processador simulat. La disminució en rendiment ve de quan fem un salt, ja que perdem un cicle on no s'executa cap operació. Pel que fa als programes "Addition" i "Extended", el primer és un programa escrit per a representar un cas òptim, on no hi ha cap salt, el programa està compost únicament per una operació de suma, que s'executa unes 600 vegades. Com podem observar, l'speedup en aquest cas és òptim, amb un valor de 2.5. Pel que fa al programa "Extended", és un programa escrit en ensamblador que executa un bucle amb 4 milions d'iteracions, on aproximadament la meitat de les operacions executades són instruccions de salt. Per tant aquest programa seria el pitjor cas possible. Com es pot observar, aquest programa ens proporciona el pitjor speedup amb diferència, però cal remarcar que, tot i així, seguim guanyant un 60% de rendiment en comparació amb la versió original.

5.2 Cicles necessaris per a executar un programa

En aquesta secció s'observa com diferents tipus de programa afecten la quantitat de cicles que tarda cada versió del simulador a executar un mateix programa. El processador original, al no utilitzar un pipeline, sempre executa una instrucció cada cicle, mentre que en la versió amb pipeline, hi haurà cicles on no executem cap instrucció perquè hem executat un salt, fent que siguin necessaris més cicles per a executar el programa. A la figura 7 podem observar fins a quin punt el tipus de programa pot afectar els cicles que tarda l'execució en el model amb pipeline.

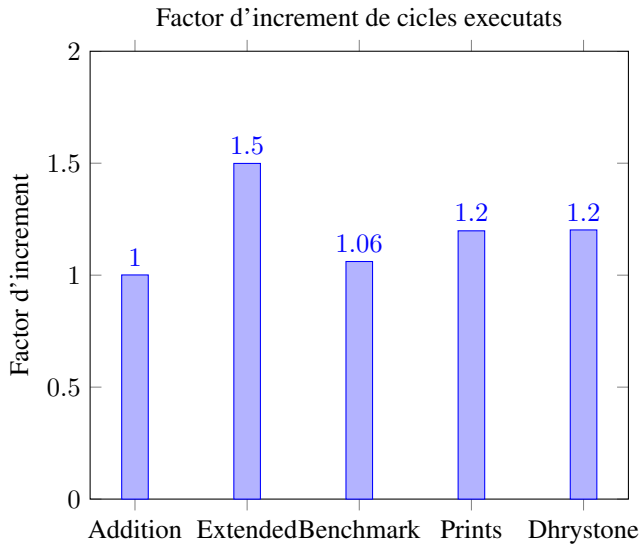

Fig. 7: Gràfic que mostra l'increment en cicles a l'executar el mateix programa amb el model de pipeline

Com es pot observar, i com al gràfic anterior, els dos primers programes ens mostren el millor i pitjor cas. En el millor cas, el factor d'increment és 1, per tant el model amb pipeline tarda els mateixos cicles que el simulador original. En el pitjor cas, podem veure que el simulador ha d'executar un 50% més de cicles que el model original, ja que en aquest programa, 1 de cada 2 instruccions són de salt, i a cada salt es perd un cicle. En la resta d'exemples, veiem que l'increment va des d'un 6% a un 20%, que són valors bastant bons.

5.3 Temps d'execució del programa

L'últim resultat que s'ha recollit és el temps d'execució de les dues versions del programa. Tot i que a nivell de simulació aquest nou model amb pipeline proporciona un gran augment de rendiment en comparació amb el model original, un aspecte important a considerar és el temps d'execució del simulador en si, ja que és el temps que realment tardarà a executar-se la simulació, i si el nou model és molt més lent que la versió original, l'experiència d'utilitzar el simulador pot empitjorar considerablement. A la figura 8 es pot observar la penalització de rendiment que comporta utilitzar la nova versió del simulador. Per a obtenir els resultats s'ha utilitzat l'eina *perf* de Linux, i el programa "Addition" no apareix al gràfic ja que la seva execució és massa curta per a obtenir resultats de temps d'execució fiables.

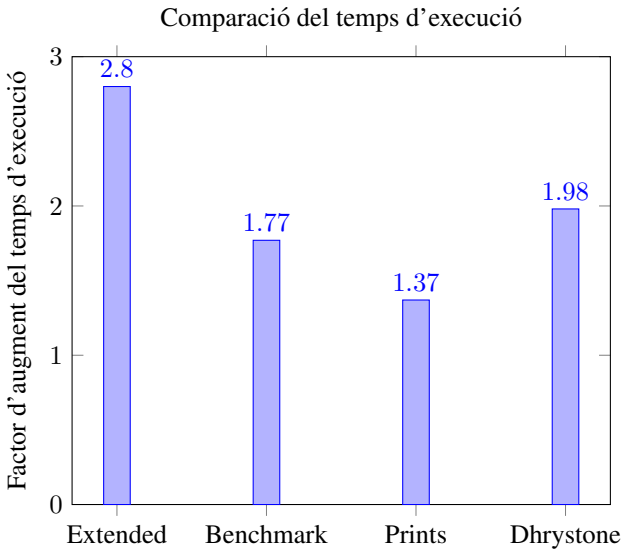

Fig. 8: Gràfic comparant la penalització en el temps d'execució entre la versió original del simulador i la versió amb execució amb pipeline

Com es pot observar a la figura 8, degut a l'overhead de gestionar les 3 etapes del pipeline i als cicles addicionals que ha de fer el processador, el temps d'execució augmenta considerablement, sobretot en el cas del programa "Extended". Tot i així es considera que l'augment de temps d'execució no afecta a la nova versió del simulador de forma suficient per a no justificar l'addició d'aquest model d'execució al simulador.

6 Conclusions

Pel que fa a aquest treball, s'han pogut completar tots els objectius amb èxit. Pel que fa al funcionament del nou model d'execució, s'ha implementat el pipeline de forma correcta, i el simulador té el mateix nivell de compatibilitat que la versió original. Pel que fa al rendiment, també s'han aconseguit bons resultats, amb un augment de rendiment de 60% en el pitjor cas, i un augment de rendiment de més del 100% en l'execució de programes convencionals. Addicionalment, també s'ha pogut veure, ni que sigui a molt alt nivell, el funcionament d'un processador RISC-V. Finalment, també existeix la possibilitat d'incloure els canvis fets al projecte original, ja que aquest és de codi obert, permetent a altres usuaris poder utilitzar aquesta versió del simulador.

7 Línies futures

Pel que fa a línies futures, hi ha quatre aspectes principals a considerar. El primer seria afegir més etapes a la simulació. Un augment del nombre d'etapes del pipeline permetria augmentar el rendiment del processador simulat encara més, ja que augmentaria el nivell de paral·lelisme. Un segon canvi a considerar seria l'execució de les etapes del pipeline en paral·lel a nivell de programa. Les etapes només s'executen en paral·lel al processador a nivell de simulació, però executar-les en diferents threads podria servir per a mitigar

la pèrdua de rendiment real quan executem el simulador. També cal considerar que l'overhead de gestionar múltiples threads podria ser superior al guany de rendiment de l'execució paral·lela, però aquesta seria una opció a considerar per a compensar la pèrdua de rendiment de la nova versió. Un altre canvi seria permetre al simulador utilitzar o bé el model amb pipeline o el model original. Aquest canvi permetria a l'usuari fer servir el model que li sigui més útil per a cada situació, ja que es podria escollir entre menor temps de simulació per a testejar software o una simulació més propera als processadors reals per a poder veure com funciona un pipeline de forma detallada. L'últim aspecte a considerar seria la compatibilitat del simulador. Tot i que el simulador pot executar programes i no presenta cap mena de problemes per a la majoria de situacions, seria interessant ampliar la funcionalitat del simulador per a assegurar el seu correcte funcionament, especialment en aquests programes menys comuns que presenten errors puntuals.

AGRAÏMENTS

M'agradaria donar les gràcies al meu tutor Màrius Montón per la seva ajuda durant el desenvolupament d'aquest treball i per proporcionar un codi base sobre el qual s'ha pogut iterar de forma senzilla. També donar les gràcies a la meva família pel seu suport durant el desenvolupament d'aquest treball i la resta del grau.

REFERÈNCIES

[1] *Lloc web de l'empresa ARM.* URL: https://www.arm.com/.

[2] *Descripció de l'arquitectura x86 (Wikipedia).* URL: https://es.wikipedia.org/wiki/X86.

[3] *Pàgina de la fundació RISC-V.* URL: https://riscv.org/.

[4] *Instruction Set Architecture (Wikipedia).* URL: https://es.wikipedia.org/wiki/Conjunto_de_instrucciones.

[5] *Reduced Instruction Set Computer (Wikipedia).* URL: https://en.wikipedia.org/wiki/Reduced_instruction_set_computer.

[6] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture.* URL: https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf.

[7] *Llicència BSD.* URL: https://opensource.org/licenses/BSD-3-Clause.

[8] *Descripció d'un pipeline.* URL: http://www2.cs.siu.edu/~cs401/Textbook/ch3.pdf.

[9] *Enllaç del simulador base.* URL: https://github.com/mariusmm/RISC-V-TLM.

[10] *Enllaç de descàrrega de la llibreria SystemC.* URL: https://accellera.org/downloads/standards/systemc.

[11] *Repositori amb la suite de proves riscv-tests.* URL: https://github.com/riscv/riscv-tests.

[12] *Repositori del simulador Spike.* URL: https://github.com/riscv/riscv-isa-sim.

[13] *Pàgina oficial del projecte QEMU.* URL: https://www.qemu.org/.

[14] *Pàgina oficial de l'empresa Imperas.* URL: http://www.imperas.com/imperas-riscv-solutions.

[15] *Pàgina oficial de FireSim.* URL: https://fires.im/.

[16] *Extensions de RISC-V.* URL: https://en.wikipedia.org/wiki/RISC-V#ISA_base_and_extensions.

[17] *Pàgina web del projecte GNU Make.* URL: https://www.gnu.org/software/make/.

[18] *Eines de RISC-V (Compilador, tests...).* URL: https://github.com/riscv/riscv-tools.

[19] *Descripció del sistema de control de versions Git.* URL: https://ca.wikipedia.org/wiki/Git.

[20] *Enllaç del repositori amb el simulador amb pipeline.* URL: https://github.com/Marc-Oros/RISC-V-TLM.