
This is the **published version** of the bachelor thesis:

Escario Becerra, Àngel. *Desenvolupament d'unitats docents basades en el processador RISC-V Wally*. Treball de Final de Grau (Universitat Autònoma de Barcelona), 2026 (Enginyeria Informàtica)

This version is available at <https://ddd.uab.cat/record/334477>

under the terms of the  license.

Desenvolupament d'unitats docents basades en el processador RISC-V Wally

Angel Escario Becerra

June 29, 2026

Resum– Aquest projecte consisteix en utilitzar un processador RISC-V Wally, per fer un ús en la docència del grau d'informàtica. El sistema permetrà al docent il·lustrar diferents aspectes del funcionament intern del processador, com per exemple els diferents estats d'execució d'una instrucció o com es recuperen dades des de la memòria en el moment que succeeix un *cache miss*. Finalment, es voldrà a desenvolupar una unitat estàndard RISC-V Wally, que es podrà utilitzar en futurs cursos per facilitar explicacions dels conceptes de l'arquitectura de computadors i creació de futurs exemples pràctics.

Paraules clau– RISC-V, ModelSim, pipeline, IEU, IFU, LSU, M-mode, IRF, caché, anàlisi, senyals, assembly, C, hardware.

Abstract– This project consists of making useful a RISC-V Wally unit for teaching purposes for the computer science degree. This system will help the teacher for showing different aspects of how the processor work internally, like showing the different execution stages of an instruction or how it is recovered the data from the memory when happens a *cache miss*. Finally, it would be likely to develop a standard unit of the RISC-V wally, which will be used for future courses, where it will be useful for teaching the concepts of the computer architecture and the creation of practical examples.

Keywords– RISC-V, ModelSim, pipeline, IEU, IFU, LSU, M-mode, IRF, cache, analysis, signals, assembly, C, hardware.



1 INTRODUCCIÓ - CONTEXT DEL TREBALL

L'expansió actual de les unitats de processament basades en RISC-V està generant una creixent demanda de professionals qualificats per part del mercat laboral. Sabent això, es pot preparar un currículum, on durant el grau es tracti amb el processador RISC-V amb una eina unificada, que pugui ser útil per diferents assignatures. Després de valorar diverses opcions *Open Source*, es va optar pel processador **RISC-V Wally**, que va ser concebut per temes docents per **D. Harris** de la **Harvey Mudd College**, i per **Sarah L. Harris** de la **University of Nevada, Las Vegas (UNLV)**. L'autor ha creat força material docent, el qual es troba disponible en forma d'unitats docents en format transparències [6] [3] i un parell de llibres [1] [2], això com d'un repositori de github [4] Altres motius per la selecció han estat:

- Naturalesa Open Source orientada a la docència: Al tractar-se d'un projecte de codi obert, permet un accés complet al codi font en llenguatges de descripció de maquinari (HDLs), facilitant que l'estudiant pugui explorar la microarquitectura real sense restriccions de llicències.
- El processador presenta un disseny modular altament parametritzable, on la configuració més bàsica comença amb l'especificació reduïda rv32e (idònia per a sistemes encastats bàsics) i es pot estendre fins a la variant d'alt rendiment rv64gc (que suporta computació de 64 bits, multiplicació per maquinari, instruccions compactes i punt flotant).
- Arquitectura segmentada de 5 etapes: Implementa la clàssica línia de producció (pipeline) de cinc estadis, ideal des del punt de vista pedagògic per analitzar fenòmens complexos de la microarquitectura com els riscos de dades (data hazards), els riscos de control (control hazards) i les tècniques de forwarding o bypass.

Abans de continuar, es vol parlar de com es podrà implementar aquest processador a nivell acadèmic en el grau

• E-mail de contacte: a.escariobecerra@gmail.com
 • Menció: Enginyeria de Computadors
 • Treball tutoritzat per: Raimon Casanova Mohr (Microelectrònica i Sistemes electrònics)
 • Curs 2025/26

d'enginyeria informàtica.

Primer, s'ha de dir que inicialment, aquest material és molt útil per entendre com pot funcionar un processador en general. A més, entendre com poder implementar configuracions pròpies, cosa que pot donar lloc a la implementació de diferents projectes o pràctiques del grau, sobretot a les assignatures relacionades amb la menció d'enginyeria de computadors. Tot això, es degut a la flexibilitat del processador, on a més d'afegir o treure mòduls, també es poden fer altres coses com arrencar un **sistema operatiu Linux** en una **FPGA**, o substituir el processador **NIOS II** que s'utilitza actualment en **Integració Hardware/Software** per aquest **RISC-V Wally**.

Després, per poder implementar aquest processador en el grau, inicialment s'haurà de fer una introducció a aquest processador en l'assignatura de **Integració Hardware/Software**, o en alguna altra assignatura prèvia, que sigui comuna per tots els alumnes matriculats en enginyeria informàtica, i en aquesta fer una introducció al processador que hauria de ser més teòrica, amb alguns exemples i fent petites proves per crear una configuració pròpia, però molt simple. Seguidament, en l'assignatura de **Integració Hardware/Software**, ja si que es podria implementar tant en pràctiques, com a exercicis de classe en diferents formats, ja que aquest processador permet ser implementat en una **FPGA**, i a classe fer exercicis relacionats amb el simulador *Questa Sim*, simulador ja utilitzat actualment en aquesta assignatura.

Un cop vist una possible opció per fer una implementació acadèmica, es veurà com en aquest document, una descripció detallada del processador **RISC-V Wally**, això és important, ja que, aquí es farà una descripció de les 5 etapes de pipeline que té aquest processador, a més dels mòduls que pot incorporar aquesta unitat de processament. Seguidament, una descripció exhaustiva de la metodologia que es seguirà en aquest treball. A continuació, es comentarà totes les proves que s'han realitzat amb aquest processador, i això inclou els possibles problemes que hagin pogut aparèixer durant aquestes implementacions, a més de les solucions que hagin sorgit durant el desenvolupament del treball. Finalment, s'analitzarà com d'útil pot ser aquesta eina, i veure si és possible que sigui implementada en les assignatures mencionades anteriorment.

2 PROCESSADOR RISC-V WALLY

El processador **RISC-V WALLY**, com s'ha comentat anteriorment, és Open Source, i amb 5 etapes de **pipeline**, que segueix les etapes clàssiques descrites en el llibre **Digital Design and Computer Architecture: RISC V Edition textbook** [1], i en **computer organization and design** [2]. Llavors, això fa que poder entendre bé el processador, en cadascuna de les 5 etapes, es pot llegir qualsevol dels dos llibres, mencionats anteriorment [1] [2]. A continuació, en la **Figura 1** es podrà veure la distribució de les 5 etapes, amb els components més importants del processador, on es pot destacar que en aquesta imatge és una vista simplificada de com es mostren les diferents parts del processador, i dels components que té com el bloc de **Multiplicació/Divisió** o **MDU** o la **Floating Point Unit**, o **FPU**.

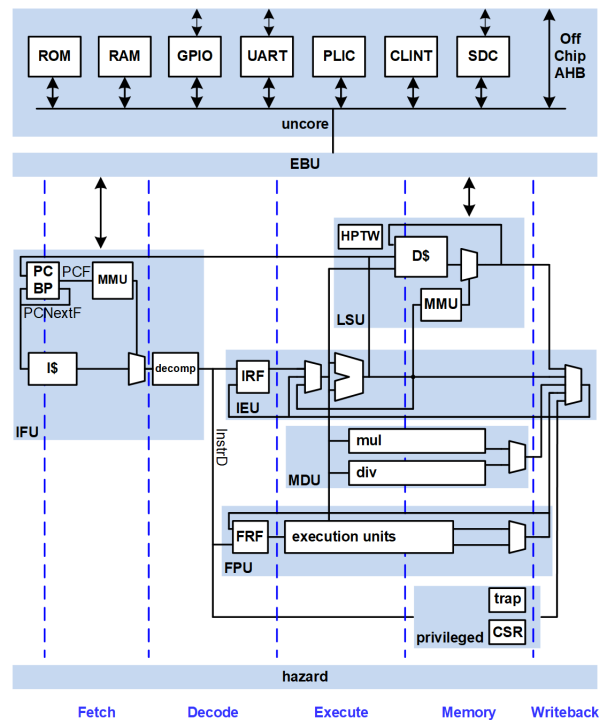


Fig. 1: Arquitectura del processador RISC-V WALLY

Analitzant la **Figura 1**, en les dues primeres etapes del processador, fetch i decode, el component més important és el bloc que constitueix la IFU (**Instruction Fetch Unit**). Aquest component és el responsable d'enviar instruccions per executar en els altres blocs del processador. Per resumir-lo, la IFU té una funció de caché que va emmagatzemant instruccions per executar.

A continuació, en l'etapa d'execució, que es pot trobar en la **Figura 1**, es troben els següents blocs: IEU (**Instruction Execute Unit**), on es troba la ALU, i que processa les instruccions descodificades en l'etapa anterior. MDU (**Multiply/Divide Unit**), s'utilitza per realitzar operacions aritmètiques de multiplicació i divisió. Per últim, està la FPU o **Floating-Point Unit**.

Seguidament, tal i com es pot veure en la **Figura 1**, en l'etapa de memòria hi ha els següents dos blocs: LSU (**Load/Store Unit**), i la **Privileged Unit**. La LSU té la funció de poder accedir a la informació de les instruccions Load i Store, i a més, llegir **page table entries** de **hardware page table walker**. En la LSU, es poden trobar diferents components com: MMU, caché, o Bus Interface. Ara en la Privileged Unit, té la funció de definir el 3 nivells generals de privilegi, que són:Usuari, Supervisor i Màquina. Els modesUsuari i Supervisor s'utilitzen per aplicacions convencionals i ús del sistema operatiu, en canvi, el mode Màquina o **M-mode**, es pot utilitzar per gestionar ambients d'execució segurs.

Per últim, en l'etapa de writeback, tal i com es mostra en la **Figura 1**, estan els següents dos blocs: IRF (**Integer Register File**), i/o FRF (**Floating-Point Register File**), que consisteix en un array de registres, que serveixen per emmagatzemar informació entre la memòria i el processador.

Un cop vist la majoria dels mòduls, una cosa que fa molt útil el processador **risc-v wally** és com molts d'aquests mòduls es poden treure o afegir a una configuració del processador, d'una manera molt senzilla, només modificant un fitxer

anomenat **config.vh**. Això, permet tenir una gran varietat de processador, on es poden tenir activat o desactivat diferents mòduls. Un exemple seria que un processador tingui un predictor de salts i una **FPU**, això ja es pot veure en les configuracions, que ja porta el processador. El més destacable seria com molts d'aquests mòduls afegeixen subconjunts d'instruccions, com per exemple operacions de punt flotant o multiplicacions i divisions.

3 OBJECTIUS

En aquest apartat es farà una descripció dels objectius que es volen assolir en el moment d'acabar aquest treball. A continuació, es farà una breu descripció dels objectius que es volen assolir durant aquest Treball de Final d'Estudis:

- Fer que es pugui mostrar per una pantalla, com es van executant instruccions d'un programa que està en C. On s'haurà de veure com una instrucció va passant per les 5 etapes del processador.
- Ser capaç de mostrar com es porten les dades des de la memòria, quan hi ha un **cache miss**, és a dir, mostrar el **cache miss**, i tot el procés que comporta portar la dada des de la memòria.
- Mostrar d'una manera clara el funcionament del branch predictor en el processador, on s'ha de poder veure en el simulador com funciona, tant quan encerta com quan falla, el predictor de salts.
- Demostrar que és possible implementar acadèmicament aquesta unitat de processament, per poder millorar l'ensenyament i donar materials de qualitat pels estudiants.

4 METODOLOGIA

En aquesta secció, es farà una descripció de la metodologia que s'utilitzarà durant aquest projecte per assolir els objectius descrits a l'apartat anterior.

4.1 Anàlisi del Repositori de GitHub

Primer de tot, i abans de començar a programar o comprovar les diferents configuracions possibles del processador, s'ha d'entendre com està estructurat el repositori de **GitHub**, on es troba tot el procés d'instal·lació en local. També, es poden trobar moltes carpetes amb molta informació, però per aquest projecte les carpetes que s'hauran de modificar durant el projecte són:

- **config**, on es troben les configuracions dels processadors, en llenguatge **Verilog**, que es podrà modificar i crear noves configuracions al gust de l'usuari.
- **examples**, aquí es crearan els diferents exemples per posar a prova i veure el funcionament dels mòduls del processador

Encara que, també hi ha altres carpetes importants que són imprescindibles per entendre bé el processador, aquestes són:

- **src**, que conté el codi font en llenguatge de descripció de maquinari (SystemVerilog) que defineix el comportament RTL del processador.
- **sim, testbench**, que són carpetes decisives per a la simulació estructural i la verificació funcional abans del desplegament.

En la carpeta **sim** es poden trobar diversos fitxers, on es poden trobar tots els fitxer relacionats en la compilació dels diferents fitxers relacionats amb les configuracions del processador. En el fitxer **wally-compile.do**, dins de la carpeta de **Questa**, es pot trobar com es compilen aquestes configuracions, degut a que el processador està descrit en llenguatge **RTL**, aquest arxiu permet compilar les configuracions que estan creades, a més de compilar les pròpies, si es van crear en el fitxer de configuracions sense fer cap modificació, en el fitxer **wally-compile.do**. En els fitxers **wally-run.do**, on aquest fitxer s'encarrega de fer la simulació. On primer, es carrega el fitxer binari del programa que s'ha compilat i executat mitjançant la comanda **vsim**. Després, utilitzant un fitxer anomenat **wave.do** que carrega totes les ones en el simulador **Questa**, i això és important, ja que són les ones que s'acaben carregant en el simulador **Questa**.

4.2 Validació dels objectius plantejats

En aquest apartat, es parlarà com validar els objectius plantejats en l'apartat 3. **Objectius**, on en cada objectiu, es farà una descripció exhaustiva de la metodologia de resolució, i arribar als resultats esperats.

4.2.1 Anàlisi de l'Estructura de Pipeline (Objectiu 1)

Per assolir aquest primer objectiu, el que es farà serà agafar un programa en C dels que ja porta el repositori de **GitHub**, i mitjançant instruccions **asm**, modificar el programa de tal manera de que es pugui carregar el valor en un registre. Després, per poder veure les 5 etapes de pipeline del processador es simularà aquest programa en el **Questa**. Seguidament, s'haurà d'analitzar el fitxer d'ones resultant. Per cercar les 5 etapes de pipeline, els creadors del **RISC-V Wally** ja han deixat un conjunt de senyals agrupades, cadascuna, en un grup que va en funció de l'etapa de pipeline del processador.

A continuació, en el cas de veure com es transfereix aquest valor que s'ha imposat en el programa en C, serà molt important analitzar el bus del processador, on s'hauria de veure com es va portant aquest operant a memòria, o a una altra variable, això serà en funció de com s'hagi programat el codi en C.

4.2.2 Avaluació del Sistema de Memòria Cache (Objectiu 2)

Després, per poder assolir el segon objectiu plantejat, es farà un programa utilitzant dos arrays que la seva mida haurà de ser superior a la cache, per poder veure com es porten les dades des de la memòria.

Per poder implementar tot això, primer s'haurà de crear un arxiu en C, que on es pugui assegurar de que hi ha errors en la cache. Després, es s'haurà de compilar el programa amb

un fitxer **Makefile**, que serà un molt similar als que hi ha per defecte, però amb les configuracions relacionades amb el programa.

Seguidament, es simularà el programa i quan es sigui possible veure el fitxer d'ones i tot s'hagi carregat, s'haurà d'afegir primer les senyals que estiguin relacionades amb la cache i la memòria, per després tornar a simular el programa.

4.2.3 Estudi del Predictor de Salts (Objectiu 3)

Llavors, per assolir el tercer objectiu proposat es reutilitzarà el programa que s'ha utilitzat per assolir el segon objectiu, on es farà alguna modificació per mostrar el funcionament del branch predictor.

Per fer això, el codi en C dissenyat per provar el predictor de salts, haurà de ser un bucle iteratiu amb instruccions condicionals dins del bucle. Amb això, es vol fer que predictor pugui cometre errors, però també encerts.

Un cop ja s'ha compilat el programa, es llençarà una simulació en el **Questa**, on l'objectiu a cercar serà trobar les instruccions condicionals, i que succeeix després d'executar aquesta instrucció, això es farà observant en la fase d'execució del processador un senyal que indiqui quina instrucció s'està executant en un determinat instant de temps.

4.2.4 Creació de configuracions

Llavors, si es volen assolir aquests objectius s'haurà de crear configuracions pròpies per a cada objectiu. Primer, el que es farà serà crear una carpeta en el directori de **config**, on es posarà com a nom el que l'usuari prefereixi, però és molt recomanable posar **rv 32** o **64**, en funció de quants bits es vol que tingui l'arquitectura, i després ficar un distintiu que ho faci diferent de les altres configuracions que hi ha per defecte. Seguidament, dins d'aquest directori creat, es ficarà un arxiu anomenat **config.vh**, on el que és recomanable seria agafar el fitxer de configuració del **rv32i** o **rv64i** ja que són els fitxer que menys coses modificades tenen respecte a les altres configuracions, i permet a l'usuari crear una gran varietat de configuracions possibles, que és una cosa que interessa molt per la docència perquè permet tant a l'alumnat com els docents plantejar diversos exercicis i llibertat per provar sense modificar el que ja s'ha creat prèviament. Un cop copiat l'arxiu en el directori de la configuració ja es podrà afegir tots els mòduls que siguin necessaris per resoldre el que s'ha plantejat. Per exemple, si es vol afegir un predictor de salts, com s'ha plantejat en l'últim objectiu, s'haurà de cercar l'apartat relacionat amb el predictor de salts, comprovar si ja està activat o no, i en cas de que no estigui, veure quins valors estan a 0 i canviar els valors a 1.

Finalment, per a cada objectiu serà prioritari crear una configuració per a cadascun dels objectius, de manera que es pugui diferenciar entre ells, com també diferenciar de les configuracions base que ja proporciona el repositori de GitHub [4].

5 PRIMERS PASSOS AMB RISC-V WALLY

A partir d'aquest apartat, es comentarà els primers experiments i intents d'aconseguir els objectius proposats a l'apartat d'**objectius**.

5.1 Arrencar RISC-V Wally

El procés de desplegament de l'entorn ha requerit la configuració del simulador Questa Sim i la clonació del repositori oficial de Wally [4]. Per a la consecució dels objectius d'aquest treball, s'ha prescindit de mòduls orientats a validacions de Linux (com QEMU o GCLIB), acotant l'entorn a la simulació estructural RTL.

Un cop s'ha fet la instal·lació, ja es pot començar a fer proves amb el simulador QUESTA SIM, o ModelSim. Aquestes proves inicials, van consistir en agafar els fitxers amb els programes en C que porta el repositori. En aquest cas s'ha agafat el programa anomenat **sum.c**, on es va compilar utilitzant el model **rv64gc**, com es pot veure en la següent imatge:

```

1 TARGET = sum
2
3 $(TARGET).objdump: $(TARGET)
4     riscv64-unknown-elf-objdump -S -D $(
5         TARGET) > $(TARGET).objdump
6
7 $(TARGET): $(TARGET).c Makefile
8     riscv64-unknown-elf-gcc -o $(TARGET) -
9         gdwarf-2 -O\
10         -march=rv64gc -mabi=lp64d -mmodel=
11         medany \
12         -nostdlib -static -lm -fno-tree-loop-
13         distribute-patterns \
14         -T../common/test.ld -I../common \
15         $(TARGET).c ../common/crt.S ../common/
16         syscalls.c
17 # Compiler flags:
18 # -o $(TARGET) defines the name of the
19 # output file
20 # -g generates debugging symbols for gdb
21 # -O turns on basic optimization; -O3 turns
22 # on heavy optimization; omit for no
23 # optimization
24 # -march=rv64gc -mabi=lp64d -mmodel=medany
25 # generates code for RV64GC with doubles
26 # and long/ptrs = 64 bits
27 # -static forces static linking (no dynamic
28 # shared libraries on bare metal)
29 # -lm links the math library if necessary (
30 # when #include math.h)
31 # -nostdlib avoids inserting standard
32 # startup files and default libraries
33 # because we are using crt.S on bare
34 # metal
35 # -fno-tree-loop-distribute-patterns turns
36 # replacing loops with memcpy/memset in
37 # the std library
38 # -T specifies the linker file
39 # -I specifies the include path (e.g. for
40 # util.h)
41 # The last line defines the C files to
42 # compile.
43 # crt.S is needed as our startup file to
44 # initialize the processor

```

```

26 # syscalls.c implements printf through
    the HTIF for Spike
27 # other flags from riscv-tests makefiles
    that don't seem to be important
28 # -ffast-math -DPREALLOCATE=1 -std=gnu99 \
29 # -fno-common -fno-builtin-printf -
    nostartfiles -lgcc \
30
31 clean:
32     rm -f $(TARGET) $(TARGET).objdump

```

Com es pot veure en el codi, es té definida com l'arquitectura ISA el rv64gc, utilitzant el model medany per a la memòria del compilador, i utilitzant lp64d per l'ABI o Application Binary Interface. Aquests factors mencionats tindran una major importància quan s'hagi d'utilitzar altres configuracions del processador.

Un cop s'ha compilat amb la comanda **make**, s'escriu la següent comanda al terminal: **wsim -sim questa rv64gc -elf sum -gui**, fa falta remarcar que sense el **-gui** no es podrà visualitzar el ModelSim. Un cop s'executa aquesta comanda s'obrirà el ModelSim, amb el waveform de l'execució del programa. Però, abans de començar a analitzar el programa s'haurà de mirar l'objdump, que conté el programa en C convertit amb instruccions asm.

00000000000001e0 <main>:

```

int main(void) {
    800001e0: 1101          addi    sp,sp,-32
    800001e2: ec06          sd     ra,24(sp)
    800001e4: e822          sd     s0,16(sp)
    int s[1], expected[1];
    setStats(1);
    800001e6: 4505          li     a0,1
    800001e8: 0b6000ef     jal    8000029e <setStats>
    s[0] = sum(4);
    800001ec: 4511          li     a0,4
    800001ee: fd9ff0ef     jal    800001c6 <sum>
    800001f2: 0005041b     sext.w s0,a0
    800001f6: c422          sw     s0,8(sp)
    setStats(0);
    800001f8: 4501          li     a0,0
    800001fa: 0a4000ef     jal    8000029e <setStats>
    printf("s = %d\n", s[0]);
    800001fe: 85a2          mv     a1,s0
    80000200: 00000517     auipc  a0,0x0
    80000204: 6d850513     addi   a0,a0,1752 # 800008d8 <atol+0x68>
    80000208: 530000ef     jal    80000738 <printf>
    int t0 = test[i], t1 = test[i+1];
    int v0 = verify[i], v1 = verify[i+1];
    if (t0 != v0) return i+1;
    if (t1 != v1) return i+2;
}
if (n % 2 != 0 && test[n-1] != verify[n-1])
    8000020c: 4522          lw     a0,8(sp)
    8000020e: 1559          addi   a0,a0,-10
    expected[0] = 10;
    return verify(1, s, expected); // 0 means success
}
    80000210: 00a03533     snez   a0,a0
    80000214: 60e2          ld     ra,24(sp)
    80000216: 6442          ld     s0,16(sp)
    80000218: 6105          addi   sp,sp,32
    8000021a: 8082          ret

```

Fig. 2: Codi asm de sum.c

En la **Figura 2**, es pot veure el main en instruccions asm. Llavors, un cop analitzat aquest objdump, es passarà a veure el waveform en el ModelSim.

En el moment de simular aquest codi mitjançant la comanda que s'ha mencionat anteriorment, es podrà analitzar les instruccions en el *Memory stage*. A més, de que les 5 etapes del processador es podran visualitzar com es van processant les instruccions del programa, i també altres components com la **Cache**, o el bus **AHB**. També, en la pàgina del Questa es poden afegir més senyals. Una cosa que s'ha de dir, i es que els creadors del **RISC-V Wally** han fet un bon treball a l'hora d'ordenar el senyals, ja que simplifica molt a l'hora de buscar les instruccions i trobar els diferents

components com la **LSU**, o la **FPU**.

6 PROVES AMB RV32I

Per provar una altra configuració, que no fos la **RV64gc**, es va optar per utilitzar el model més senzill, el **RV32I**. Aquest processador no té ni memòria RAM, o memòria ROM com a components, ja que, la RAM està incorporada dins de la **LSU** o Load-Store Unit, i la ROM es troba dins de la IFU o **Instruction Fetch Unit**. Això es pot veure millor en el fitxer que inclou el DTIM, anomenat dtim.sv.

Per fer les proves es van fer alguns programes simples en C, i els més simples, utilitzant instruccions asm mitjançant la comanda asm que C ens permet utilitzar. En la següent imatge tenim un programa en C que utilitza la funció asm per carregar un valor en un registre.

Després, es crearà el makefile per poder compilar aquest programa, on s'utilitzarà el rv32i-zicsr, s'explicarà en el següent apartat de perquè això i no rv32i. A més, es fan canvis al mabi, utilitzant ilp32, i al mcmmodel, que serà med-low.

Seguidament, utilitzem la comanda **make**, per compilar el programa.

Listing 1: Objectdump per carregar un registre en RISC-V

```

1 int main() {
2 80000208: ff010113          addi    sp,
    sp,-16
3     volatile int result;
4     // Standard RAM address for many RISC-V
    systems
5     int *target_address = (int *)0x80002000
    ;
6     // RISC-V Inline assembly for 'lw' (
    Load Word)
7     __asm__ __volatile__ (
8
9 8000020c: 800027b7          lui     a5,0
    x80002
10 80000210: 0007a783          lw     a5,0(a5
    ) # 80002000 <tohost>
11 80000214: 00f12623          sw     a5,12(
    sp)
12     : "r" (result)          // Output:
    result goes into a register
13     : "r" (target_address) // Input:
    address goes into a register
14
15 );
16
17     return 0;
18 80000218: 00000513          li     a0,0
19 8000021c: 01010113          addi    sp,
    sp,16
20 80000220: 00008067          ret
}

```

Un cop s'ha compilat es llença el programa amb la comanda **wsim -sim questa rv32i -elf tester -gui**. Finalment, es podrà veure com es van executant les instruccions del main en el waveform. On l'objectiu serà veure com es carrega el valor en un registre.

6.1 Problemes amb el rv32i

Per poder fer tot això que s'ha explicat a l'apartat anterior, es va tenir que fer prova i error, on l'error més greu va ser que quan s'executava la comanda **wsim -sim questa rv32i -elf tester -gui** s'obria el Questa Sim, però no podia carregar cap mòdul i sempre donava error, i al moment de finalitzar la carrega dels mòduls el Questa es tancava, llavors no donava temps a saber que era el que fallava exactament. Però, el problema es trobava en que no podia carregar les instruccions CSR o **Control and Status Register**, aquestes instruccions serveixen per llegir, modificar o escriure en els registres de **control i status** [5]. Un cop es va incloure al *Makefile*, el programa ja no donava cap problema i el waveform s'obria, i es podia començar a analitzar l'execució del programa.

Per dur a terme les verificacions descrites en la secció anterior, es va procedir a un procés d'assaig i validació experimental de la plataforma. Durant aquesta fase, es va identificar una incidència crítica en la inicialització de l'entorn de simulació: en executar l'ordre de llançament **wsim -sim questa rv32i -elf tester -gui**, la interfície gràfica de Questa Sim s'iniciava correctament, però es mostrava incapacitada per demanar i carregar els mòduls estructurals del processador. Aquest comportament provocava una fallada seqüencial immediata que forçava el tancament abrupte del simulador de manera instantània, fet que impedia la lectura i la traçabilitat directa dels registres de diagnòstic d'error en el terminal de treball.

Després d'una anàlisi detallada del flux de compilació, es va determinar que l'origen de la incompatibilitat radicava en l'absència del suport actiu per a les instruccions de tipus **CSR (Control and Status Registers)**. Aquests registres compleixen una funció essencial en l'arquitectura RISC-V, ja que s'encarreguen de la lectura, modificació i escriptura de l'estat intern i dels privilegis del processador [5] [7]. Una vegada es va integrar la regla de dependència corresponent i la inclusió d'aquests registres dins del fitxer *Makefile*, el sistema es va compilar sense anomalies. Aquesta correcció va habilitar finalment el desplegament del cronograma de senyals (**waveform**), permetent iniciar l'anàlisi sistemàtica i la verificació funcional de l'execució del programa.

6.2 Proves per analitzar el mòdul de multiplicació/divisió (MDU)

Un cop es va veure que es podia executar amb un altre model de processador, es va fer una segona prova per veure com s'anava executant una instrucció, i es va fer en el següent programa en C:

Listing 2: Simulació de multiplicació en RISC-V

```

1 #include <stdio.h>
2
3 extern volatile long long tohost;
4
5 int main(void) {
6     volatile int A[10] = {0, 1, 2, 3, 4, 5,
7                           6, 7, 8, 9};
8     volatile int B[10] = {2, 4, 6, 8, 10,
9                           12, 14, 16, 18, 20};
10    volatile int C[10];

```

```

10    for (int i = 0; i < 10; i++) {
11        C[i] = A[i] * B[i];
12    }
13
14    // Keep these to control the simulation
15    tohost = 1;
16    while(1);
17    return 0;
18 }

```

Aquest programa fa una multiplicació entre dos arrays. Per poder compilar aquest programa es va utilitzar un makefile similar a l'anterior, però amb la diferència de que el valor del paràmetre *march* era **rv32im-zicsr**, on la m carregava el mòdul de multiplicació, ja que el rv32i no ho porta per defecte.

Un cop s'ha compilat, s'executa, i s'obre el waveform, on es pot identificar la instrucció **MUL**, que ens indica que en aquest moment s'està executant la instrucció de multiplicació. Aquesta instrucció és senzilla de detectar, degut a que en la configuració està activat el mòdul de **Multiplicació/Divisió**, a més de com s'ha programat, per part dels desenvolupadors, fa que es pugui observar fàcilment com s'executa una instrucció de multiplicació, i identificar-la correctament en el fitxer d'ones. Una recomanació per a que surti així, és la d'imposar que totes les variables siguin **volatile**, d'aquesta manera s'obliga al compilador a executar-les.

7 PRIMERS PASSOS A CREAR UN NOU PROCESSADOR

Després de fer les proves que s'han mencionat anteriorment, es va decidir crear una nova configuració de processador. Es va escollir el **rv32i** com a base, i a partir d'això incloure alguns components que no té com el **bus, RAM o ROM**. Es farà una copia del directori on es troba **rv32i**, i es canviarà el seu nom, en aquest cas serà **rv32i AWC**, que fa referència a que aquesta configuració tindrà cache. A més, s'incorporen les *CSR instructions*, ja que així no donarà problemes a l'hora d'executar la comanda **wsim**. Una cosa molt bona que tenen aquests fitxers de configuració, es que només s'ha de canviar el valor d'una variable per poder activar certs components com la RAM. A més, només s'han de fer modificacions en el fitxer de configuració, ja que en el següent arxiu anomenat **wally.do**, es pot trobar com es carrega la carpeta de configuracions, tal i com es mostra en la **Figura 4**.

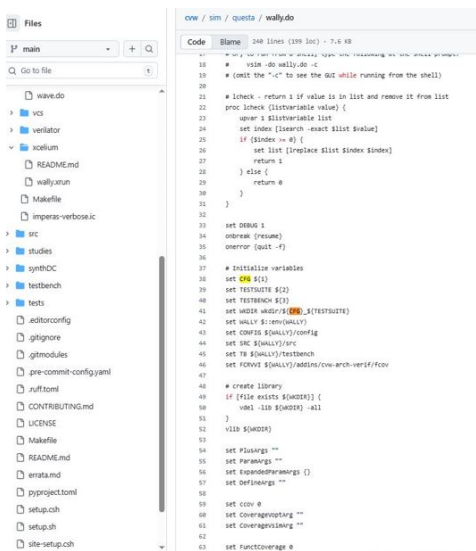


Fig. 4: Arxiu wally.do

Listing 3: Configuració de paràmetres del sistema

```

1 // LSU microarchitectural Features
2 localparam logic BUS_SUPPORTED = 1;
3 localparam logic DCACHE_SUPPORTED = 1;
4 localparam logic ICACHE_SUPPORTED = 1;
5 localparam logic
6     VECTORED_INTERRUPTS_SUPPORTED = 1;
7 localparam logic BIGENDIAN_SUPPORTED = 0;
8
9 // Hardware performance counters
10 localparam logic ZICNTR_SUPPORTED = 1;
11 localparam logic ZIHPM_SUPPORTED = 1;
12 localparam COUNTERS = 12'd29;
13
14 // Cache-management operation extensions
15 localparam logic ZICBOM_SUPPORTED = 1;
16 localparam logic ZICBOZ_SUPPORTED = 1;
17 localparam logic ZICBOP_SUPPORTED = 1;
18
19 localparam logic ZICSR_SUPPORTED = 1; //
    CSR Instructions

```

8 IMPLEMENTACIÓ I PROVES AMB EL PROCESSADOR IMPLEMENTAT RV32I_AWC

8.1 Implementació

Per poder implementar el processador *rv32i_AWC*, s'haurà de realitzar primer alguns canvis, ja que es volen implementar diverses millores respecte al processador *rv32i*, ja que com s'ha comentat anteriorment, es vol incorporar una cache, on a més s'hauran d'incorporar més elements. Per fer això, es va agafar el fitxer de configuració del processador *rv32i*, ja que és el més senzill, i per tant, el més personalitzable.

Per poder aplicar tots els canvis s'haurà de canviar alguns valors que es troben en el fitxer *config.vh*, i es posarà a 1 els valors que s'han decidit prèviament, on les següents variables passaran a tindre el valor a 1:

8.2 Proves amb el processador rv32i_AWC

Per començar amb les proves, es va realitzar una prova simple que també es va realitzar amb el processador *rv32i*, és la de carregar una valor mitjançant la comanda *asm*. Després, s'executa mitjançant la comanda **wsim -sim questa rv32i_AWC -elf tester -gui**, per poder veure la simulació en el Questa Sim.

Els resultats obtinguts d'aquesta simulació, es podrà veure un cop hagi finalitzat la simulació, com es carrega a memòria el valor que es va indicar en el programa, on es pot a veure a més, com passa pel bus *AHB*. També, en l'altra finestra que s'obre quan s'executa la comanda per simular, es poden trobar més senyals relacionades amb el bus *AHB* o el component *lsu*.

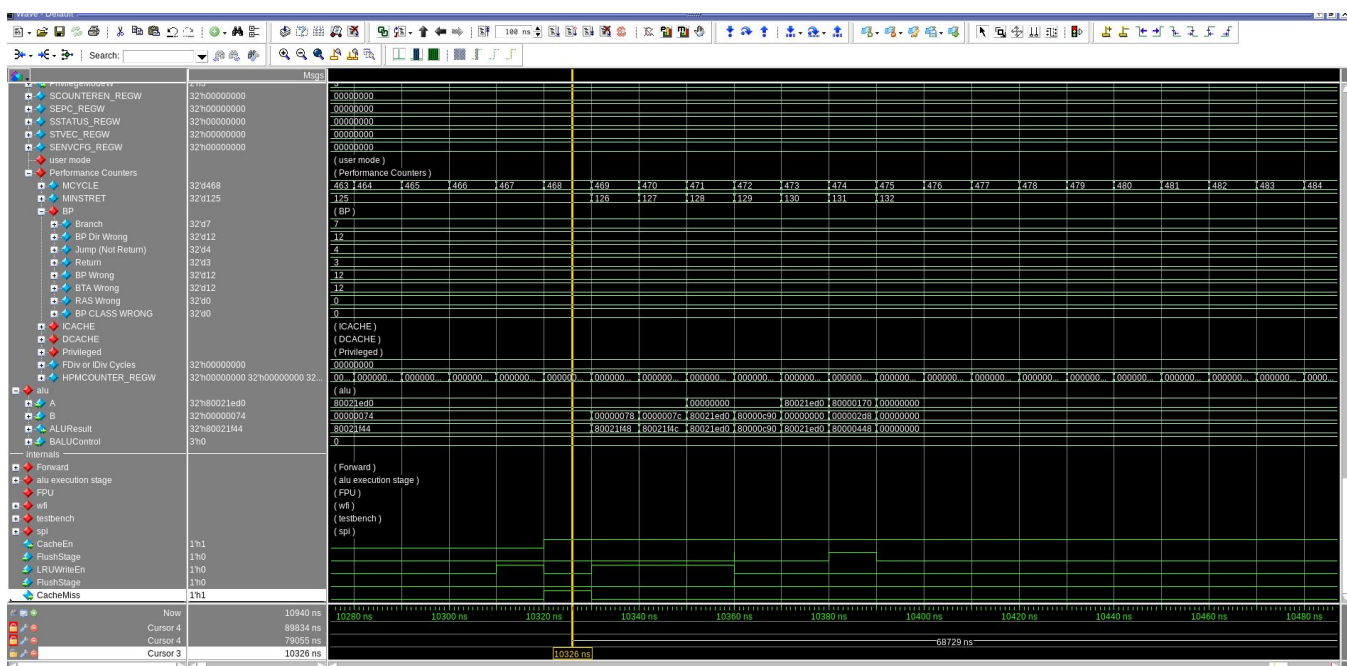


Fig. 3: CacheMiss trobat durant la simulació

Per la segona prova, es va voler veure els *Cache miss* que podien succeir durant l'execució del programa en la simulació. Primer, quan es va obrir el programa per primer cop, s'havien d'afegir els senyals **Hit** i **CacheMiss**. Llavors, analitzant la **Figura 3**, es recompila i es torna a simular per poder obtenir els senyals afegits, tot això es fa des del Questa. El senyal **Hit** es troba ja inclosa en la simulació, i es troba dins de la **LSU**, dins de **dcache**. Però, per trobar i afegir la senyal **CacheMiss**, s'ha d'anar al Questa i fer el següent: **lsu, bus, dcache, dcache**, i en aquest últim s'haurà de cercar el senyal **CacheMiss**.

Un cop ja afegides els senyals a la simulació, es pot veure com es van produint els errors de cache i es van mostrant durant tota la simulació.

Però, una cosa que es va observar al programa es que tardava massa en fer la simulació degut a que les multiplicacions es desconstruïen en sumes, i això, dificultava trobar les operacions que estaven dins del bucle intern. Llavors, es va optar per crear una nova versió del processador que tindrà inclòs el mòdul de **Multiplicacions i Divisions**.

8.3 Problemes amb rv32i_AWC

Durant la fase de compilació de la variant **rv32i_AWC**, es va identificar un conflicte de parametrització en el fitxer `config.vh`. Concretament, l'activació de les extensions de comptadors de rendiment (**ZICNTR** i **ZIHPM**) requereix definir explícitament el rang del paràmetre **COUNTERS** (fixat en **12'd29**) [8]. Sense aquesta definició, el compilador genera un error sintàctic que impedeix l'elaboració del directori de la configuració.

Listing 4: Configuració del paràmetre Hardware performance counters

```
1 // Hardware performance counters
2 localparam logic ZICNTR_SUPPORTED = 1;
3 localparam logic ZIHPM_SUPPORTED = 1;
4 localparam COUNTERS = 12'd29;
```

Un cop el codi queda com el que hi ha a sobre, definint els comptadors, ja no hi ha cap problema i es pot executar qualsevol programa.

Aquests comptadors de hardware són uns registres de propòsit especial, que es dediquen a comptar el temps que tarda en executar-se una instrucció.

9 IMPLEMENTACIÓ I PROBLEMES AMB EL PROCESSADOR RV32I_AM

9.1 Implementació

Per implementar aquesta nova versió del processador es va agafar el fitxer de configuració del **rv32i_AWC**, i es va activar el mòdul de multiplicació, és a dir, es va posar a 1 el valor del paràmetre del bloc de multiplicació.

Listing 5: Configuració de paràmetres del sistema per activar el mòdul de multiplicació

```
1 // Multiplication & division extensions
2 // M implies (and in the configuration file
   requires) Zmmul
3 localparam logic M_SUPPORTED      = 1;
4 localparam logic ZMMUL_SUPPORTED = 1;
```

9.2 Problemes amb rv32i_AM

La integració del mòdul de multiplicació i divisió (**extensió M**) en el nucli **RV32I** va presentar incompatibilitats en la síntesi de la **unitat de divisió (MDU)**. Atesa la naturalesa docent del projecte i per evitar penalitzacions en els temps d'execució, es va optar per acotar el disseny a una configuració purament de sumes lligades, posposant la resolució d'aquest mòdul per a línies d'investigació futures.

10 BRANCH PREDICTOR

Per poder comprovar el funcionament i veure com es pot aplicar a la docència es va optar per comprovar-ho amb dues configuracions diferents: la primera, no té incorporada una cache, en canvi en la segona, sí que està implementada, tant la **ICACHE**, com la **DCACHE**.

10.1 Branch predictor sense cache

Per poder implementar aquesta configuració del processador *risc-v Wally*, només s'haurà d'activar les instruccions relacionades amb el branch predictor, on s'ha agafat el fitxer de configuració del processador **rv32i**.

Listing 6: Configuració de paràmetres del sistema per activar el mòdul de branch predictor

```
1 localparam logic BPRED_SUPPORTED = 1;
2 localparam INSTR_CLASS_PRED = 1;
```

Un cop s'ha personalitzat la configuració del processador, ja es pot passar a simular programes en el **Questa Sim**. El primer programa, consisteix que en funció del valor de la variable **i**, on després es passa per una condició de si el nombre és parell o no, retornarà un resultat i ho sumarà a un acumulador.

Per veure com el predictor ha comès un error, s'ha d'identificar que just després de que s'executi una instrucció de tipus *Branch*, en aquest cas **BNE** per exemple, s'executa una instrucció de tipus **NOP/FLUSH**. Aquest tipus d'instrucció, indica que s'ha tingut que netejar les instruccions que hi havien previstes per executar. En canvi, quan el predictor encerta es pot veure com just després no hi ha una instrucció de tipus **NOP/FLUSH**, si no que segueix l'execució que s'havia proposat.

En el segon programa, consisteix en un programa molt similar a l'anterior, amb la diferència de que es fan dos if, i d'aquesta manera es vol veure si el branch predictor, i el processador poden entendre que només fa falta entrar dins d'un de les dues condicions.

Per l'últim programa a simular, on es vol provar com funciona el branch predictor quan hi ha un patró pseudoaleatori de condicions, on es comprova si el valor del vector és 1 o diferent. D'aquesta manera és molt més difícil trobar un patró per anar repetint els salts.

Aquests errors en el predictor de salts que es generen en la simulació són deguts en la seva majoria a que el predictor decideix a no fer el salt, llavors segueix executant instruccions, però després ha d'executar una instrucció de tipus **NOP/FLUSH**, ja que en molts casos, el predictor ha fallat i té que netejar el pipeline d'instruccions que tenia planejat a executar.

10.2 Branch predictor amb cache

En aquest apartat es compararà l'execució del mateixos programes, però ara, utilitzant una nova configuració que incorpora una cache. Això és important, perquè s'ha inclòs la cache d'instruccions. A més, es vol també comprovar que la penalització per tenir un error en la predicció del salt és del doble, ja que es pot tenir un error en la predicció del salt, i a més, també tenir un *cache miss*. El major canvi que es pot veure a l'hora d'executar els programes, és quan succeïx un error degut a que el predictor ha errat en la seva predicció, com es pot analitzar en la **Figura 5**, i a més hi ha un error en la cache, això fa que la penalització sigui el doble que sense cache, ja que sense es va per defecte a la memòria.

Per poder veure els errors que ha fet el predictor de salts, és tan senzill com anar a veure com es van executant les instruccions, i veure com després d'una instrucció de tipus *Branch* com *BEQ* o *JAL*, s'executa una instrucció *NOP/FLUSH*, cosa que es pot analitzar observant el senyal **InstrEName** en el grup d'**Execution Stage** en la **Figura 5**. Ja que, s'han d'eliminar les instruccions del pipeline que s'executaran posteriorment després del branch. En canvi, per veure els *Cache misses* s'haurà de veure que en el moment de fer un *NOP/FLUSH* s'ha de veure en el bus **AHB** hi ha un *burst* de 16 cicles de rellotge, però això dependrà de la mida de la cache.

11 CONCLUSIONS

En conclusió, primer de tot és un treball amb una finalitat molt clara i un objectiu que és assolible. Encara que, hi ha feina que fer, i primer de tot, s'haurà de començar a analitzar programes senzills per poder entendre bé el funcionament del processador Wally. On, un cop entès aquest funcionament, es podrà a començar a seguir els passos definits anteriorment per poder assolir els objectius definits en aquest apartat.

Durant la realització d'aquest treball s'ha pogut investigar molt sobre el funcionament del processador per posar exemples pràctics, que després d'aquest projecte, es podrien posar en pràctica ja en el programa, per quan passi el canvi del pla acadèmic, aquest processador o ja estigui implementat o que la seva implementació estigui molt avançada. Tot això és important perquè és una tecnologia *open source*, i permetria a la universitat col·laborar amb altres universitats en projectes relacionats amb aquest processador, i convertir a la **UAB** en una potència en aquest processador.

Finalment, es vol dir que aquest treball ha sigut molt interessant, i que pot ajudar a millorar la docència del grau, a més de que qualsevol alumne es pugui ficar a treballar en aquest processador pel seu compte, per poder proposar noves maneres d'implementar aquest processador. A més, de que aquest treball pot ajudar a que sigui com una base per investigacions futures, sobretot investigació acadèmica.

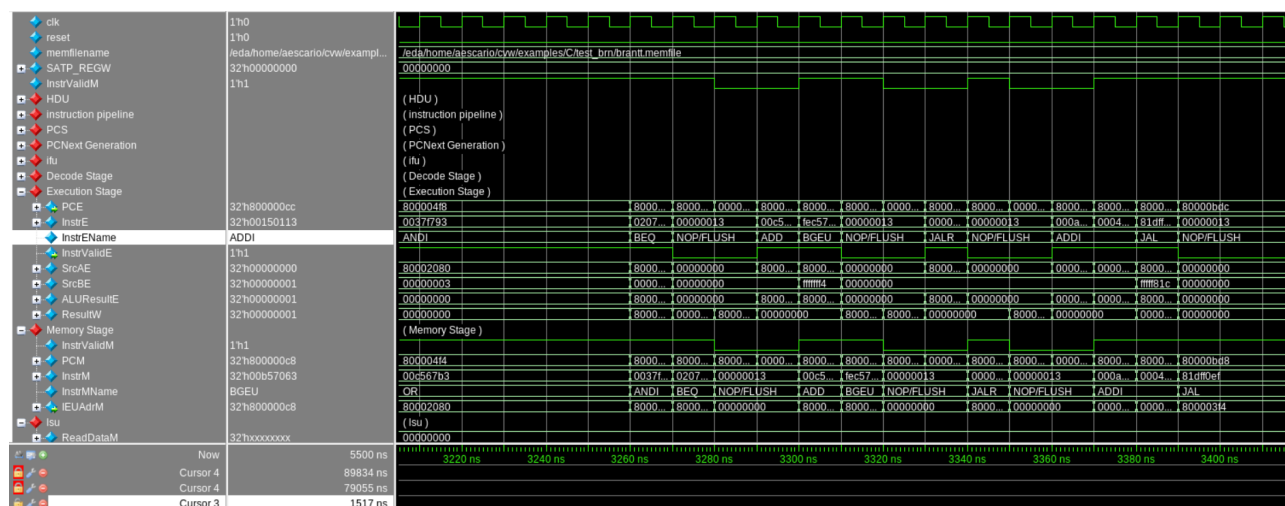


Fig. 5: Doble penalització degut al cache miss i predictor

ÚS DE LA IA GENERATIVA

S'ha fet ús de la Intel·ligència Artificial per la correcció del text, per saber com solucionar un problema relacionat amb latex, i per últim, a cercar algunes fonts, que posteriorment s'han revisat. També s'ha utilitzat la IA per resoldre alguns problemes que es podien trobar en el fitxer *Makefile*, i de arreglar alguns programes en C.

AGRAÏMENTS

Voldria agrair a Raimon Casanova Mohr per donar-me la oportunitat de fer aquest TFG, i a openhwgroup per haver creat el processador Wally amb el repositori de GitHub [2].

REFERENCES

- [1] http://www.r-5.org/files/books/computers/hw-layers/hardware/digital-design/David_Harris_Sarah_Harris-Digital_Design_and_Computer_Architecture-EN.pdf
- [2] <https://books.google.com.bo/books?id=U6lpXmu7OngC&printsec=copyright&hl=es#v=onepage&q&f=false>
- [3] <https://web.fdi.ucm.es/posgrado/conferencias/SarahHarris-slides.pdf>
- [4] <https://github.com/openhwgroup/cvw/tree/main>
- [5] <https://book.rvemu.app/hardware-components/03-csrs.html>
- [6] <https://drive.google.com/drive/folders/126yYoF6p0vFH-qdP-bMw4utHXb8u2iqX>
- [7] <https://docs.riscv.org/reference/isa/priv/priv-csrs.html>
- [8] https://en.wikipedia.org/wiki/Hardware_performance_counter