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

Oduardo Castillo, Marcos. *Lockspike : Desenvolupament d'un framework de verificació 'lock-step' respecte a Spike per a un simulador RISC-V64GC*. Treball de Final de Grau (Universitat Autònoma de Barcelona), 2026 (Enginyeria Informàtica)

This version is available at <https://ddd.uab.cat/record/334366>

under the terms of the  license.

Lockspike: Desenvolupament d'un framework de verificació lock-step respecte a Spike per a un simulador RISC-V64GC

Marcos Oduardo Castillo

30 de juny de 2026

Resum— La verificació del correcte funcionament d'una CPU és una tasca complexa, ja que implica comprovar que el seu comportament sigui exactament igual al descrit per l'especificació del sistema en qualsevol dels casos que aquesta contempla. Un mètode de verificació modern en la indústria és la simulació *lock-step*, que consisteix a executar de forma paral·lela un simulador *golden model* amb el device under test instrucció a instrucció. No hi ha cap eina estàndard per instrumentar Spike, el simulador referència del ISA de RISC-V, conjuntament amb un altre simulador, especialment en Python donat que el cas d'ús típic és la verificació d'un disseny en un llenguatge de descripció de *hardware*. Aquest treball pretén desenvolupar el *framework* de simulació *lock-step* per al simulador *Punxa* per completar l'objectiu de corregir els errors existents en el seu disseny fins aconseguir carregar una versió minimalista del kernel de Linux.

Paraules clau— CPU, *lock-step*, Device Under Test (DUT), Instruction Set Architecture (ISA), especificació, divergència, *bug*, verificació, *framework*.

Abstract—The verification of the correct functionality of any CPU is a complex task, as it requires ensuring that the code behavior is equal to that defined by the specification of the system in every context it defines. One modern verification method used by the industry is the lock-step verification, which consists in executing in parallel a golden model, typically a simulator, with the device under test instruction by instruction. There isn't a standardized tool to instrument Spike, the reference simulator of the RISC-V ISA, in conjunction with another simulator, specially in Python given that the typical use case is the verification of a design made in a hardware description language. This work aims to develop a lock-step simulation framework for the Punxa simulator in order to achieve the objective of fixing the existing errors in its design until it is capable of booting a minimal version of the Linux kernel.

Keywords— CPU, lock-step, Device Under Test (DUT), Instruction Set Architecture (ISA), specification, divergence, bug, verification, framework.

1 INTRODUCCIÓ - CONTEXT DEL TREBALL

LA verificació del disseny d'un processador es tracta d'un procés complex a causa de la interpretació de l'especificació del sistema creat. L'especificació d'una CPU es tracta de la definició del conjunt

de comportaments que ha de realitzar un processador per catalogar-se com a complidor del repertori d'instruccions [11]. El problema és que no hi ha una forma directa de validar tots els comportaments d'un sistema: normalment s'han de preparar diversos tests amb resultats definits per poder comprovar-los amb les respostes que genera el sistema a verificar [4], i molts cops aquests tests poden ometre certs *edge cases* de l'especificació i mai no reportar errors en l'execució d'aquestes instruccions mal dissenyades. El problema s'incrementa quan vols validar un sistema que no només simula una CPU, sinó un *System on Chip* sencer. En augmentar la complexitat es necessiten tests més extensos que puguin validar tots els casos possibles i la verificació

-
- E-mail de contacte: marcos.oduardo@gmail.com
 - Menció: Enginyeria de Computadors
 - Treball tutoritzat per: David Castells Rufas (Departament de microelectrònica i sistemes electrònics)
 - Curs 2025/26

cada cop es torna molt més difícil de configurar. En aquest context es troba Punxa [9], un simulador comportamental de RISCv64GC que vol validar el seu comportament del ISA i dels seus perifèrics fins arribar a carregar el kernel de Linux. El problema és que tot i passar els tests individuals de cada instrucció correctament, el sistema es troba amb kernel panics al tractar de bootar una versió reduïda del kernel. Per això s'ha optat per executar Punxa pas a pas amb un model de referència, el simulador *golden model* Spike [3], que actua com a guia i serveix per verificar cada instrucció executada entre els dos sistemes, i si Punxa té un valor que divergeix respecte el de Spike significa que la implementació és errònia. Aquesta metodologia és coneguda en la indústria com a *lock-step verification*. Tot i així, les eines disponibles per realitzar aquest procés no són adequades per l'arquitectura de Punxa: Els devices under test típics solen ser sistemes descrits en llenguatges HDL, i per això els *frameworks* de verificació no s'integren bé amb les necessitats de Punxa, escrit en un llenguatge d'alt nivell [1] [2]. Per aquesta raó s'ha decidit confeccionar un *framework* de verificació *lock-step* en Python que permeti instanciar l'estat de la simulació de Spike per el valor dels seus registres de propòsit general, CSRs, memòria, perifèrics necessaris per la validació del device under test i que s'encarregui de sincronitzar les diferències arquitectòniques dels dos simuladors, amb l'objectiu final d'aconseguir carregar el prompt de login a la consola del simulador. Aquest *framework* de verificació té el nom de Lockspike [14]. A més, per la realització d'aquest treball s'ha de construir una versió de Linux suficientment minimalista perquè la seva execució sigui factible en l'entorn del simulador. Per aconseguir aquest objectiu, s'han de realitzar diverses tasques que componen el gruix del treball. Les seccions posteriors detallen:

1. La planificació del projecte, amb les tasques necessàries que s'han realitzat en el seu desenvolupament i la seva planificació temporal.
2. Els objectius que marquen la completació de les tasques anteriorment enumerades.
3. Explicació de l'arquitectura del mecanisme de simulació *lock-step*.
4. Explicacions del desenvolupament de cada tasca o funcionalitat, amb les seves consideracions individuals.
5. Metodologia utilitzada per coordinar el desenvolupament del projecte.
6. Divergències trobades amb Lockspike, les seves causes i les seves solucions.
7. Exposició dels resultats del projecte en relació amb els objectius plantejats inicialment i conclusions extretes del treball realitzat.

2 PLANIFICACIÓ DEL PROJECTE

2.1 Tasques del projecte

Tasca	Setmanes
1. Instrumentació de Spike	1–4
2. Imatge Linux minimalista	4–5
3. Arrencada Linux en Spike	5–6
4. Validació de viabilitat	6
5. Comparació d'estats	6–8
6. Diagnosi i correcció del boot de OpenSBI	8–11
7. Creació de <i>checkpointing</i> per a Spike	11–13
8. Diagnosi i correcció del boot de Linux	11–16
9. Càrrega de Linux en Punxa	17

La planificació del projecte contempla les 9 tasques relacionades amb el procés de verificació del simulador indicades a la taula 1. El desenvolupament del projecte ha seguit un model d'organització de tasques majoritàriament seqüencials, a excepció de la tasca 7 que va ser realitzada en paral·lel amb la tasca 8. Inicialment, no es contemplava la creació de la tasca 7, però la seva necessitat va aparèixer en progressar cap a les fases posteriors al boot del *firmware* d'OpenSBI [8]. Sense una funcionalitat de *snapshot* de l'estat de Spike, la verificació pot arribar a dificultar-se pels alts temps d'execució fins trobar la pròxima divergència. A més, es tracta d'una funcionalitat que es pot contribuir a la comunitat open-source i que més desenvolupadors podrien trobar útil per verificar els seus propis sistemes. La planificació temporal es va fer computant unes 20 setmanes de treball abans de l'entrega final. El compliment de l'objectiu es va marcar i complir a la setmana 17, deixant 3 setmanes de marge per tractar modificacions temporals a la planificació i com a contingència per arribar a l'objectiu proposat, a més del temps necessari per redactar la documentació del projecte. Les explicacions del procés de desenvolupament de cada tasca i la seva necessitat es detallen en futures seccions.

2.2 Objectius del projecte

Els objectius del projecte són els 7 enumerats a continuació:

- Instrumentar Spike des de Python
- Generar la imatge de Linux minimalista per a RV64GC (Buildroot)
- Verificar que Linux arrenca en Spike instrumentat
- Implementar el *testbench* de simulació *lock-step*
- Implementar la funció de *checkpointing* a Spike
- Diagnosticar i corregir els errors de Punxa
- Càrrega completa de Linux en Punxa

Cada objectiu representa una fita realitzada que permet al projecte seguir desenvolupant-se, i que en funció de la seva completació s'avaluaran diferents conclusions en l'avaluació dels resultats. L'objectiu final, definit com 'càrrega completa de Linux en Punxa' es dona per vàlid en aconseguir arribar al *prompt* de *login* del *shell* d'usuari. Arribar a complir aquest objectiu significa que el procés de verificació de Punxa ha sigut un èxit i que el *framework*

TAULA 1: TASQUES DEL PROJECTE

pot ser utilitzat per validar diferents sistemes executant diferents programes, donat que siguin configurats els mecanismes adequats per a la sincronització de les diferències arquitectòniques amb Spike, els quals no poden ser configurats de manera genèrica entre DUTs.

3 SCRIPT DE VERIFICACIÓ LOCK-STEP

La verificació *lock-step* és un mecanisme de verificació que consisteix en l'execució pas a pas d'instruccions en dos simuladors diferents, comparant els efectes de les instruccions en l'estat de la simulació i verificant la seva correcta implementació. La idea principal és que si dos simuladors del mateix ISA, amb els mateixos devices respectant l'especificació teòricament aconseguirien executar qualsevol binari de forma adequada si els dos es comporten correctament.

El *framework* de verificació *lock-step* confeccionat actua com un agent exterior al context de la simulació dels dos sistemes, que orquestra l'execució del simulador a verificar i Spike com a *golden model*. És un *script* en Python que s'encarrega de carregar el context comú inicial dels dos simuladors. Inicialment, tots dos tenen els seus estats individuals i els seus propis devices, tot i que caldrà sincronitzar certs comportaments asíncrons i diferències arquitectòniques per aconseguir una verificació factible. La figura 1 mostra un esquema del funcionament del *framework* respecte els dos simuladors.

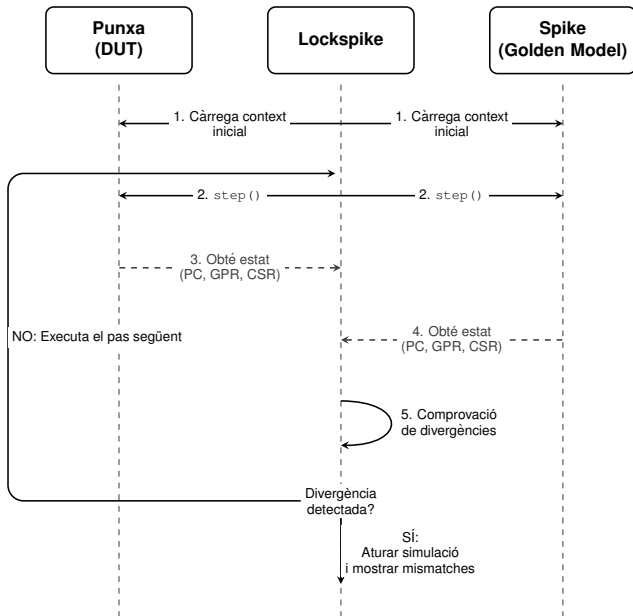


Fig. 1: Diagrama de seqüència simplificat de l'orquestració externa en *lock-step*.

Es pot observar que l'*script* és el que instància els dos simuladors individualment per executar instruccions pas a pas. La principal complexitat és que els mecanismes de simulació *lock-step* són intrínsecament dependents a l'arquitectura del DUT, i cal sincronitzar certs aspectes del sistema que causen divergències, no per implementacions errònies sinó per mecanismes que segons l'especificació tenen llibertat per variar segons la implementació (notòriament: modes de paginació suportats, interrupcions, extensions del ISA...etc). Al moment de la publicació d'aquest article, tot i existir diverses discussions sobre el tema [5] no existeix

cap interfície estandarditzada per verificar amb *lock-step*, i les implementacions han de ser molt personalitzades amb llibreries externes a les que defineix Spike. A més, tampoc existeix un mètode generalitzat per instanciar una simulació de Spike i poder manipular el seu estat des d'un *script* extern (específicament en Python en aquest cas), tot i existir diversos *frameworks* que habiliten aquestes funcionalitats. Les següents tasques del procés de desenvolupament justifiquen la seva designació en necessitats per aconseguir realitzar el procés de verificació.

4 DESENVOLUPAMENT DEL PROJECTE

4.1 Instrumentació de Spike

Aquesta tasca consistia a trobar un mecanisme per poder dirigir una instància de simulació de Spike i consultar el seu estat des de Python per poder organitzar el *framework* de simulació *lock-step*. En la investigació es va trobar un *framework* anomenat "Hammer" [13] que proveeix una infraestructura C++ i Python per interactuar amb Spike i dirigir el seu comportament. Tot i així, al final es va decidir utilitzar el *framework* PySpike [6] per realitzar el projecte. PySpike és un projecte que s'encarrega d'exposar els *bindings* de Spike cap a una interfície invocable des de Python. Els *bindings* disponibles permeten modificar l'estat del PC, registres de propòsit general, instanciar el processador, generar la configuració del mapa de memòria i definir el ISA entre altres operacions. PySpike vendoritza la seva pròpia versió de Spike modificada, i aquesta és la raó que va convertir-la en l'opció més adient, ja que la seva interfície per configurar els *bindings* és simple, podem realitzar els canvis necessaris al codi font de Spike respecte a les nostres necessitats i el repositori té suport del mantenidor actualment, la qual cosa significa que els artefactes desenvolupats podran ser retornats com a ampliacions del projecte perquè més desenvolupadors aprofitin el nou *framework*. Hammer, tot i que també fa servir la mateixa llibreria que PySpike per exposar els *bindings* de Spike i el seu *framework* ja habilita la instanciació de Spike pas a pas des de python, el seu repositori està arxivat i la forma que té de vendoritzar el seu propi Spike configurat és amb un *patch* file, molt més difícil de configurar per a les nostres necessitats. Per aquestes raons es va decidir desenvolupar PySpike com a base del *framework lock-step*.

En Lockspike instanciem la classe *sim* amb el seu mètode *step*, que és el mateix que fa servir spike per avançar tot el sistema en funció de l'execució d'una instrucció. Per aconseguir instanciar el mètode *step*, que és privat al codi original, vam haver de modificar la visibilitat del mètode en la classe a *public* i afegir una definició per exportar la funció cap a Python en PySpike. A més, també es van exposar mètodes per realitzar loads i stores a l'espai de memòria de Spike. D'aquesta manera podem carregar els binaris a executar necessaris, i altres continguts importants com el *device tree*. Finalment, va sorgir la necessitat de crear un mecanisme per simular la *bootROM* de spike i carregar un *device tree* personalitzat, degut a que en el codi font Spike no deixa accedir als continguts de la *bootROM* com una regió de memòria més (la protegeix contra escriptures), i spike carrega el *device tree* des d'un argument en la crida del seu executable. Com nosaltres instanciem directament

la classe *sim*, hem de realitzar aquesta configuració manualment: inicialitzem la simulació i l'espai de memòria amb el seu objecte de configuració *cfg*, modifiquem el PC de la simulació per apuntar a una adreça de memòria amb una *bootROM* personalitzada, indiquem l'adreça on carregarem el *device tree* en el registre *a1* i saltem a l'adreça d'OpenSBI. En aquest moment, tenim una instància de la simulació de Spike manipulable des de Python en àrees de contingut de registres, PC, i continguts de memòria.

4.2 Creació d'una imatge de Linux minimalista

La creació de la imatge de Linux que carregarem en la memòria dels dos simuladors ha de ser minimalista, amb els continguts mínims perquè la seva càrrega sigui eficient i lleugera. S'ha de tenir en compte que Punxa és un simulador en Python: això vol dir que el seu temps d'execució és substancialment lent, i per això no podem agafar qualsevol imatge convencional sinó una dissenyada per a sistemes encastats de baixos recursos. L'objectiu principal és aconseguir arribar al prompt de login del sistema, així que s'ha de compilar la imatge de linux amb un entorn *initramfs* que permeti interactuar amb el sistema en la fase de l'script d'init. Per confeccionar aquest *initramfs* es va decidir utilitzar BusyBox amb musl per aconseguir un *userspace* funcional però eficient i factible de carregar en Punxa. Aquesta imatge de Linux minimalista es configura i compila amb l'eina de Buildroot [7], específicament una configuració denominada *tinyconfig* que enllaça només les funcionalitats més bàsiques i indispensables del sistema. Aquesta imatge de Linux es compila dins d'una imatge d'OpenSBI, específicament la versió que inclou el kernel com un *payload* que carrega en memòria. La imatge resultant del *firmware* del sistema (OpenSBI + kernel + *initramfs*) ocupa aproximadament 27MB.

4.3 Arrencada de Linux en el Spike Instrumentat i validació de viabilitat

Després de comprovar que la imatge de Linux està correctament configurada i carrega en l'executable de Spike oficial, cal comprovar que la càrrega és factible en l'entorn de simulació *lock-step*. Els únics canvis que es van haver de realitzar van ser configuracions a les extensions indicades a l'entorn de compilació del kernel, ja que el Spike instrumentat que fem servir ha de suportar exactament les mateixes extensions que Punxa. El temps total que triga en carregar Linux és considerablement més lent que la versió nativa en C++ (com es detalla a la Taula 2 amb la comparativa del *slowdown*), però igualment triga segons en arribar al prompt de login i el sistema és totalment funcional. Aquest punt marca la validació de viabilitat de PySpike com a base del *framework* de simulació *lock-step*. Com hem aconseguit carregar Linux en l'entorn de Lockspike i tenim accés a gran part de l'estat de simulació de Spike amb una àmplia capacitat de personalització de les capacitats del *framework*, es pot considerar viable l'enfocament del projecte i no cal avaluar altres alternatives.

4.4 Mecanisme de detecció de divergències

Aquest és el component principal de Lockspike que habilita la verificació dels simuladors. El *framework* té una funció principal d'execució pas a pas. En cada pas, executa la instrucció actual en els dos simuladors i recull el seu estat (PC, GPRs, CSR determinats). A continuació, itera sobre cadascun dels registres i determina si hi ha cap divergència. En finalitzar la comparació, continua amb l'execució de la simulació si l'estat dels dos simuladors respecte els registres inspeccionats és equivalent. Si no ho és, avorta l'execució de les instruccions i mostra les diferències trobades en els registres determinats. Aquest mecanisme incorpora un *ring buffer* perquè, al moment de trobar una divergència i aprofitant funcions del desensamblador de Punxa, mostra les 10 últimes instruccions executades abans del *mismatch* per poder determinar quina és la causa de la divergència. Aquesta és la versió més bàsica del mecanisme *lock-step* i es va desenvolupar per aconseguir verificar l'estat en el boot del *firmware* d'OpenSBI fins al salt a l'adreça on està carregat el kernel. Aquest mecanisme ha patit diverses transformacions detallades en tasques posteriors per poder sincronitzar correctament els comportaments que depenen de l'arquitectura del simulador, com les interrupcions, fins a la càrrega completa del sistema operatiu.

4.5 Diagnosi i correcció del boot de OpenSBI

En aquesta tasca s'aplica el mecanisme de detecció de divergències desenvolupat per validar Punxa fins al final del boot del *firmware* OpenSBI. Cal notar que no tots els errors trobats impossibiliten tant la càrrega d'OpenSBI com de Linux, però provoquen divergències davant d'una versió verificada del mateix sistema. Per validar els comportaments discutits a continuació s'ha fet servir l'especificació oficial de RISC-V per a les instruccions privilegiades [11]. A continuació, es llisten les divergències trobades amb el *framework* i les seves solucions:

- a) **SLTIU bug:** La implementació de Punxa de la instrucció SLTIU inicialitza el registre de destinació a 0 abans de fer l'operació de comparació entre el registre d'origen i el valor literal indicat. Si el registre de destinació és el mateix registre que el d'origen, la comparació sempre es fa contra el valor 0, retornant un resultat potencialment incorrecte.
- b) **Multiplexed Bus bug:** Una operació decremental incorrecta per avaluar si l'adreça d'un accés pertany a un determinat *slave device* del sistema (CLINT, UART, etc.) donava un fals positiu, a causa del fet que la substracció de l'offset només s'havia de realitzar un sol cop. La solució va ser convertir l'operació en un càlcul aïllat en cada iteració, en comptes de decrementar la variable. També es va afegir un *break* al moment de trobar el dispositiu al qual pertany l'accés, per evitar possibles coincidències addicionals i sobreescriptures dels valors.
- c) **pmpaddr bug:** Els CSR *pmpaddr* del 16 fins al 63 no estan implementats, ni els CSR *pmpcfg* utilitzats per definir la seva disponibilitat. Segons l'especificació privilegiada de RISC-V, tots els registres *pmpcfg* i *pmpaddr* han de ser accessibles, i fins i tot els no implementats

han de poder rebre escriptures de qualsevol valor i retornar 0. La solució és implementar-los en el codi de Punxa per evitar un *trap*, i seguidament realitzar una comprovació per determinar si es tracta d'un dels CSR no implementats i retornar 0.

- d) **Instruction access fault bug:** En determinades instàncies (per exemple, executant un CSR no implementat) el simulador de Punxa emetia un error per Instruction Access Fault. Aquest error és incorrecte en aquestes situacions, atès que l'IAF només ha de ser emès si hi ha qualsevol problema a l'hora d'accedir a la instrucció, no d'executar-la. La solució és reemplaçar aquesta excepció per un *trap* d'Illegal Instruction, adequat per a aquests casos.
- e) **stimecmp bug:** Els CSR *stimecmp* i *stimecmph* no estan implementats a Spike, però en Punxa sí. La solució és comentar-los en el codi font de Punxa perquè ambdós simuladors generin un *trap* d'Illegal Instruction en accedir-hi.
- f) **mstatus dirty bits bug:** El CSR *mstatus* ha de posar el seu bit més significatiu (SD, bit 63) a 0 o a 1 depenent de si els registres de *floating-point* o els *extended registers* han estat modificats o no, per mantenir un control sobre quins registres han de guardar el seu estat en el *context switch*. Això es controla observant l'estat dels bits 16:15 (XS) i 14:13 (FS): si estan a 11, llavors han estat modificats. Punxa no implementava aquest comportament en la lectura de *mstatus*.
- g) **tinfo bug:** El CSR 0x7A4 (*tinfo*) no està suportat a Punxa, però a Spike sí. La solució és implementar-lo en la llista de CSR de Punxa amb el valor corresponent.
- h) **tselect triggers bug:** El CSR *tselect* només implementava un *trigger* a Punxa, i a més simplement ignorava les escriptures, de manera que no permetia seleccionar cap *trigger*. En Spike s'implementen 4 *triggers*. La solució és implementar correctament els *triggers* i la condició per seleccionar-los.

Després de resoldre aquestes divergències, OpenSBI està validat fins al salt a l'adreça del kernel.

4.6 Creació de *checkpoints* per spike

Aquesta tasca es va realitzar en paral·lel al procés de verificació de Linux en Punxa, i neix com una necessitat creada per l'arquitectura de Punxa, amb l'objectiu d'accelerar el procés iteratiu en l'entorn d'execució del kernel. El problema principal és el temps necessari per trobar una divergència: En versions anteriors sense optimitzacions de l'script de verificació, executant-se en un processador Intel(R) Core(TM) i5-8350U CPU @ 1.70GHz solament el procés de càrrega d'OpenSBI, que són aproximadament 5M d'instruccions, podia arribar a trigar 40min en temps d'execució. La càrrega total del sistema operatiu es realitza en aproximadament 110M instruccions. Per verificar els punts finals de la inicialització del kernel, tornar a executar 6h d'instruccions ja verificades és un cost molt alt i un mecanisme relativament ineficient si el nombre de divergències és molt gran o si els fixes que es proven resulten no ser

la solució correcta teoritzada. La solució és implementar un mecanisme de *checkpoints* per Spike, de forma que es pugui guardar l'estat de la simulació cada cert temps i estalviar hores de temps d'execució. Punxa ja implementa la seva pròpia funció de *checkpointing*. La idea és enfocar els *checkpoints* com una eina creada amb la base de PySpike, d'aquesta forma es tracta d'una funcionalitat totalment separada del *framework* de verificació *lock-step* i podria ser contribuït al projecte de PySpike. La idea és que cada 10M instruccions podem realitzar un *checkpoint* automàtic en els dos simuladors des del *framework*, i en cas de trobar una divergència, aplicar la solució i tornar a carregar el *checkpoint* més proper. El mecanisme de *checkpoint* ha de respectar les limitacions arquitectòniques que fixa Spike en el seu funcionament: hi han certs CSR als que no es poden accedir dos cops (els de 64 bits conformats per 2 registres diferents), i la creació i restauració dels *checkpoints* s'ha de realitzar en M-mode per poder accedir a les regions protegides per la PMP. A més, per a la restauració dels CSR de la FPU s'ha de verificar que la unitat de FPU està activa. Provisionalment el format dels *checkpoints* de Spike és el mateix que el dels de Punxa; Es pot crear un *checkpoint* amb Spike i carregar-lo en Punxa, i el mateix a la inversa. El mecanisme de *checkpointing* serialitza PC, GPR, FPR, regions de memòria, estat del CLINT (*mtime*, *mtimecmp*) i mode de privilegi. Cal notar que certes configuracions del kernel es realitzen en runtime tenint en compte característiques dels devices l'estat dels quals no se serialitzen als *checkpoints*, per tant, el comportament del sistema al carregar els *checkpoints* pot no ser l'esperat.

4.7 Diagnosi i correcció del boot de Linux

Aquesta és la part que més temps del projecte ha consumit, degut a l'elevat nombre d'instruccions a verificar, les diferències arquitectòniques entre els dos simuladors i els diferents mecanismes que s'inicialitzen i es configuren en el runtime del boot de Linux.

En el context d'execució del kernel, cal diferenciar 2 tipus principals de divergències que veuen les seves causes per motius principals diferents, els quals han sigut catalogats com:

cosim-bound: Sorgeixen per les diferències arquitectòniques dels dos simuladors. Divergir no significa que hi ha una implementació errònia en el DUT, l'especificació pot no imposar certs comportaments i deixar-los com a *implementation defined behaviors*: Els sistemes tenen llibertat a escollir com implementar els comportaments, normalment dins d'un cert marge de coherència. En cas de divergir per aquesta causa, cal determinar un mecanisme de sincronització on un dels simuladors imposa el seu comportament respecte a l'altre. Un exemple d'aquests casos són les interrupcions, asíncrones per naturalesa i que poden succeir en diversos moments de l'execució de la instrucció.

DUT-bound: Aquestes divergències són causades per implementacions diferents entre els dos sistemes. En aquest cas, si comptem amb Spike com a *golden model* amb el comportament ideal, les diferències en la simulació són causades per un comportament incorrecte que s'ha de solucionar. Aquests són els coneguts com a *bugs*. Per validar els comportaments discutits a continuació s'ha fet servir l'especificació oficial de RISC-V per a les instruccions privilegi-

ades [11], i el interrupt cookbook [12] de SiFive per trobar informació sobre el comportament de les interrupcions i els devices que les configuren (PLIC, CLINT).

A continuació, es mostren les divergències DUT-bound trobades:

- a) **STIP set bug:** El mecanisme per activar STIP en Punxa era incorrecte segons l'especificació [11]. Si la CPU no té l'extensió Sstc les interrupcions de *timer* no poden ser delegades per l'arquitectura. El CLINT, quan $mtime \geq mtimecmp$, activa el bit MTIP del CSR MIP i agafa el *timer interrupt* en M-mode, llavors si el bit per delegar *timer interrupts* està actiu (bit 5 de MIDELEG) escriu al bit STIP del registre MIP en l'*exception handler*. Punxa feia el *set* de STIP manualment si detectava que MIDELEG estava actiu per a les interrupcions de *kernel*, fent que el simulador no agafés la interrupció en M-mode sinó en S-mode. El *set* de STIP ha de ser realitzat per OpenSBI, no pel *hardware*.
- b) **MIE interrupt bug:** La implementació en Punxa de les condicions per agafar una interrupció eren errònies. El codi en Punxa només prenia interrupcions M-mode si el bit MIE estava activat. Això és incorrecte, atès que en S-mode les interrupcions M-mode han d'agafar-se sempre tot i que MIE no estigui actiu. La lògica és la mateixa per a les interrupcions S-mode en U-mode.
- c) **UART implementation:** La implementació en Punxa del model de la UART estava incompleta: Actuava com una interfície "tonta" que imprimia per pantalla qualsevol escriptura que rebia a la seva adreça de la línia Tx. En una implementació real, existeixen diversos registres multiplexats que realitzen diferents configuracions al device, i Punxa no les implementava. Per aquesta raó el sistema funcionava de forma errònia en assumir funcionalitats de l'UART inexistents. La solució és implementar el model funcional de l'UART de Punxa complet.
- d) **SUM bit in paging:** La lògica per permetre l'accés al kernel a certes pàgines d'U-mode era incorrecta. Punxa no permetia l'accés al kernel a cap pàgina d'usuari, però l'especificació indica que si el bit SUM en sstatus està configurat, el kernel pot accedir a aquesta pàgina. Aquest error causava un kernel panic en intentar executar les primeres instruccions de l'script d'init. La solució és configurar correctament la comprovació de permisos per les pàgines.
- e) **FPU dirty status set:** La lògica per configurar l'estat dels registres de la FPU en mstatus no es realitzava per a certes instruccions que interactuaven amb aquesta unitat. La solució és detectar en l'etapa d'execució si la instrucció modifica l'estat de la FPU i realitzar els canvis pertinents als CSR.
- f) **ASID implementation mismatch:** El CSR satp té certs bits que tenen com a comportament WARL. Si el sistema operatiu escriu a satp, llegeix els valors i veu que el resultat de la lectura no posa a 0 certs bits del registre assumeix que el sistema implementa ASID, un mecanisme per codificar la propietat de certes entrades de la TLB a diferents processos i no haver de realitzar un flush de la TLB en cada canvi de context del sistema. Punxa no implementava ASID i Spike sí, i Linux operava amb la suposició de que els sistemes el feien servir. El resultat és que en certes instruccions de lectura després d'un canvi de context Punxa llegia dades brusa i Spike no. La solució va ser implementar correctament ASID en Punxa. Si Spike no hagués implementat l'ASID, una altra solució seria codificar correctament el masking a 0 dels bits que comproven l'ASID a Punxa.

A continuació, les divergències *cosim-bound* trobades:

- a) **Mode de paginació:** El DTB que fem servir és generat per Spike. Aquest DTB té suport a diferents modes de paginació, però Punxa només té suport per a sv39. Cal declarar únicament aquest mode al DTB per evitar divergències entre els dos simuladors.
- b) **INTERLEAVE yield reservation:** Al codi font de Spike existeix una variable de valor fix denominada INTERLEAVE, amb valor 5000. Es compara el nombre de passos de simulació realitzats amb aquest valor, i si és igual o superior es realitzen una sèrie de tasques: canviar el *hart* a avançar, es realitzen els *ticks* corresponents a cada *device* extern (CLINT) i s'anul·la la reserva generada per instruccions *Load Reserved*. Això causa una divergència de comportament entre Punxa i Spike en invalidar certes operacions en moments diferents. La solució és eliminar la línia d'anul·lació de reserva a Spike per mantenir el comportament equivalent als dos simuladors.
- c) **Timer interrupt mismatch:** És la part més complexa de simular en *lock-step* a causa de l'asincronia dels esdeveniments. Les interrupcions, a diferència dels *traps*, poden succeir en qualsevol moment de l'execució, i no està determinat en l'especificació per la seva pròpia naturalesa. Punxa i Spike tenen diferents formes de processar els *ticks* en cada *device* (en Punxa els dispositius processen les pulsacions individualment en cada *clk* i en Spike es fa en *batches* de 5000 passos, no hi ha *clocks* formals per al processament d'instruccions ni es compten cicles del *pipeline* d'execució). Per a aquest cas, hem de definir que un dels dos simuladors serà el que defineix els temps de les interrupcions, notificant a l'altre al moment d'agafar-les. En cada pas sincronitzem el *mtime* de Punxa al mateix valor que el de Spike, al mateix temps desactivem el registre *mtimecmp* de Spike per evitar que pugui disparar cap *interrupt* pel seu compte. Cada cop que Punxa rep una interrupció, l'*script* de simulació la detecta i força $mtimecmp = mtime$ en Spike per activar el bit de MTIP i realitzar la interrupció.
- d) **PLIC external interrupt mismatch:** Els dos sistemes, Punxa i Spike, tenen un PLIC per configurar interrupcions externes dels dispositius. La UART estava configurada al *device tree* per enrutar les seves interrupcions a la CPU mitjançant el PLIC. Tot i així, el *framework* no té un mecanisme per sincronitzar les interrupcions externes. Com les interrupcions externes generaven divergències tot i eliminar l'enrutament de les interrupcions de l'UART, es va optar per eliminar el PLIC del *device tree*, així que resta inutilitzat en Punxa i

Spike. La UART en aquesta nova configuració funciona mitjançant *polling* i timer interrupts.

- e) **WFI implementation mismatch:** L'especificació no obliga a adoptar una implementació formal de la instrucció WFI, que posa al sistema en repòs mentre espera una interrupció. Spike la modela correctament, però Punxa l'ignora. La solució pot ser que Spike l'ignori temporalment, o que Punxa l'implementi igual que Spike. No ha fet falta realitzar aquesta correcció per aconseguir l'objectiu del treball, donat que es tracta de l'última divergència.

4.8 Càrrega de Linux en Punxa

Un cop resoltes les divergències anteriorment esmentades, Punxa aconsegueix arribar a carregar el kernel, executar un programa en *userspace* (init) amb el kernel proporcionant serveis de paginació, protecció de memòria, execució de *syscalls* i canvis de context, i finalment mostra el *prompt* de *login* de Buildroot per executar un *shell* interactiu dins de *l'initramfs*, amb Busybox servint eines comunes com *cat*, *ls*, *cd* i *grep* en un sol binari. Aquest punt marca l'assoliment de l'objectiu final del treball, i demostra que el *framework* de verificació *lock-step* implementat ha sigut un èxit i s'ha aconseguit solucionar tots els errors que Punxa contenia en el seu disseny i li impedié que carregués el sistema operatiu.

5 METODOLOGIA IMPLEMENTADA

La metodologia implementada originalment consisteix en aquest procés iteratiu de 4 passos:

- Executar la simulació *lock-step* fins a la primera divergència.
- Analitzar la instrucció divergent i determinar-ne la causa.
- Corregir el codi font de Punxa.
- Tornar al pas a) i verificar que la correcció permet avançar en la seqüència d'arrencada de forma esperada.

El seguiment del treball es realitzava mitjançant reunions setmanals amb el tutor on es decidien les pròximes tasques a realitzar segons l'estat del desenvolupament del projecte.

Atès que en el procés de desenvolupament del treball s'han trobat diversos reptes, no necessàriament vinculats amb el correcte funcionament de Punxa sinó amb la plataforma de simulació *lock-step*, una versió més fidel de la metodologia implementada és la següent:

- Executar la simulació *lock-step* fins a la primera divergència.
- Analitzar la instrucció divergent i determinar-ne la causa. Típicament 2 tipus: *cosim-bound* i *DUT-bound*.
- Si és el cas *DUT-bound*, corregir el codi font de Punxa. Si és el cas *cosim-bound*, determinar el simulador que dictarà l'estat corresponent de la causa de la divergència i sincronitzar.

- Tornar al pas a) i verificar que la correcció permet avançar en la seqüència d'arrencada de forma esperada.

El mètode de seguiment del treball ha sigut mitjançant revisions periòdiques amb el tutor segons s'aconsegueixen completar els objectius o es requereix centralitzar l'atenció del desenvolupament en noves funcionalitats on s'exposen els avenços realitzats amb el projecte i s'exploren diferents modificacions a realitzar.

6 RESULTATS DEL DESENVOLUPAMENT DEL PROJECTE

En aquesta secció s'exposen els resultats obtinguts en el desenvolupament del projecte, en relació amb els objectius marcats.

TAULA 2: COMPARATIVA DE TEMPS D'ARRENCADA D'OPENSBI FINS A L'ENTRADA DEL KERNEL (0x80200000)

Plataforma	Temps (s)	Factor
Spike (C++ natiu)	0.138	1.0×
PySpike (C++ pas a pas des de Python)	9.200	66.6×
Punxa	711.74	5157.5×
'Lock-step' Punxa + PySpike	1453.52	10532.7×

En la taula 3 es poden observar les comparatives del temps d'execució del boot d'OpenSBI per a les diferents plataformes testejades. Els resultats més significatius són:

- Iterar Spike pas a pas des de Python amb els *bindings* exposats per PySpike genera un alentiment d'un factor de 66,6.
- Punxa és un simulador amb components del sistema modelats amb la llibreria *py4hw* [10], desenvolupat íntegrament en Python. El comportament dels dispositius modelats simula la propagació dels senyals a nivell de cicles de rellotge, tot i ser una versió *single-cycle* del processador. El llenguatge utilitzat i l'arquitectura del sistema provoquen que Punxa sigui 5.157,5 vegades més lent que el simulador Spike.
- La velocitat de Punxa és el principal coll d'ampolla de l'execució de la simulació *lock-step*. A més, s'observa que el bucle principal de revisió de l'estat alenteix l'execució en un factor de 2,04, essencialment doblant-lo.
- És assumible que, si haguéssim de verificar un altre simulador dissenyat en un llenguatge de programació amb temps d'execució més eficients, la sobrecàrrega (*overhead*) de la verificació es mantindria constant, pròxima a un factor de x2.

Respecte als resultats extrets del desenvolupament i l'aplicació de Lockspike, podem extreure que:

- S'ha utilitzat Lockspike per identificar un total de 14 errors d'implementació en el comportament de Punxa. Aquests errors impedié carregar el sistema operatiu Linux. Amb la seva correcció, el sistema aconsegueix carregar i executar-se correctament.

- b) S'ha hagut d'adaptar Lockspike per sincronitzar un total de 6 divergències causades per l'arquitectura dels simuladors.

Lockspike té un mecanisme per verificar les escriptures a memòria i la seva equivalència en els dos sistemes. El funcionament total d'aquest sistema no ha sigut verificat totalment en el desenvolupament del projecte: només ha sigut utilitzat per concloure que la causa de la divergència de dues lectures a memòria incorrectes no era la d'una escriptura no realitzada per part de Punxa (sent la causa real el *bug* de l'ASID). A més, la implementació d'aquesta verificació requereix implementar certs *hooks* en el simulador DUT, i augmenta la complexitat d'implementar el *framework* amb altres DUTs que no siguin Punxa.

A més, el *framework* té certs punts *cecs* que no verifica en runtime: Determinats valors dels CSRs que no són comparats com a divergències, no es comparen els estats dels registres de la FPU ni dels seus CSR, i les operacions amb memòria són difícils de verificar sense interfícies que exposa el DUT.

El factor més significatiu que influència l'estratègia de verificació és la dificultat de determinar quin és l'error a partir del *report* dels registres divergents i les 10 últimes instruccions executades. A diferència d'altres *frameworks* de tests unitaris individuals, el boot no realitza cap indicació de quin és el procediment que està realitzant o quina part de l'especificació és la que dictamina el comportament realitzat. Tantmateix, amb la informació proporcionada del context d'execució del *framework* implementat ha sigut suficient per corregir els errors que causaven el crash del kernel. És considerable que aquesta dificultat és el preu a pagar per realitzar aquesta estratègia de verificació, que pot exposar els comportaments més complexos i específics per executar correctament un binari de forma més directa que qualsevol altre test.

7 CONCLUSIONS

S'ha arribat a una sèrie de conclusions en la realització d'aquest treball: La primera de totes és que queda demostrat que es pot instanciar i controlar el simulador de Spike a voluntat amb les modificacions afegides al *framework* de PySpike. A més, aquesta instanciació i les modificacions afegides faciliten l'ampliació de funcionalitats i eines desenvolupades per a entorns de simulació (p. ex. l'eina de *checkpointing* desenvolupada) les quals poden beneficiar la comunitat global de desenvolupadors per a tasques com la realitzada en aquest treball. Respecte la simulació *lock-step* entre els simuladors Spike i Punxa (i en conseqüència, altres simuladors), s'ha aconseguit realitzar instrumentant Spike amb el *framework* modificat de PySpike. Les diferències arquitectòniques entre els dos simuladors poden ser identificades i mitigades per assolir una sincronització correcta d'esdeveniments intrínsecament asíncrons, i amb aquestes sincronitzacions l'estratègia de *lock-step* és efectiva per trobar errors en el comportament dels simuladors a un nivell que engloba tota l'arquitectura del sistema, a diferència d'altres tipus de test que se centren en comportaments específics del sistema simulat (p. ex. els tests de RISC-V, que passaven correctament a Punxa però en el moment d'executar la verificació es troben errors en el comportament

respecte els privilegis, o fins i tot en instruccions amb uns inputs i outputs específics o en extensions de signe). Tot i així, el mètode de verificació amb *lock-step* no és infalible: existeixen certes àrees de comportament (revisió dels continguts de memòria, comportament de la MMU, registres de la FPU) que la solució desenvolupada no revisa i podrien dificultar la verificació del simulador (però els seus efectes segueixen sent observables a l'estat de la simulació, de manera que el sistema actual tard o d'hora els detecta tot i que la causa pugui no semblar evident). Això demostra que el mecanisme de verificació de *lock-step* no és superior ni inferior a altres mètodes utilitzats a la indústria com els tests unitaris o la verificació amb *instruction fuzzing*, sinó que és una eina més de l'ecosistema que s'adequa com a solució ideal segons les necessitats de cada projecte. En el cas d'aquest treball, el mecanisme de simulació *lock-step* es tracta de la solució ideal per aconseguir l'objectiu de verificar Punxa fins carregar Linux de forma efectiva i directa. Respecte als efectes de l'entorn de *lock-step* al rendiment de la simulació, l'*overhead* es calcula en aproximadament un factor de 2 en el temps d'execució respecte el d'executar Punxa sol, i aquest *performance overhead* és assumible en el context del propi rendiment del simulador en Python. Finalment, el mecanisme de simulació *lock-step* ha estat utilitzat de forma satisfactòria per trobar i solucionar els *bugs* arquitecturals de Punxa que impediè el correcte boot de Linux. Els 7 objectius designats al començament del treball han sigut completats seguint la metodologia descrita per aquest treball.

8 PRÒXIMS PASSOS

El projecte no queda tancat, donat que encara es pot expandir en diferents punts del seu desenvolupament. Ara que hem aconseguit verificar Punxa fins arribar al boot de Linux, podem fer servir la implementació corregida de Punxa i el *framework* de Lockspike per verificar la versió multicicle de Punxa. També es pot treballar a realitzar la contribució de la funcionalitat de *checkpoints* i els *bindings* modificats al projecte de PySpike, per tal que altres desenvolupadors puguin utilitzar el simulador Spike segons les seves conveniències. A més, el projecte de Lockspike segueix sent desenvolupat com a projecte open-source. Qualsevol desenvolupador que vulgui implementar una estratègia de verificació *lock-step* amb el seu propi DUT pot agafar Lockspike com a base i adaptar el *framework* per sincronitzar les diferències arquitectòniques dels seus propis simuladors amb les de Spike, o simplement buscar el test d'instanciació individual a Spike des de Python i utilitzar Spike segons les seves necessitats. Tot i que el *framework* és funcional i en relació a l'objectiu d'aconseguir *checkpoints* per spike, algunes característiques com la serialització de certs devices de Spike (UART, PLIC) en els *checkpoints* no estan actualment realitzades, i, sobre l'objectiu d'instrumentar Spike des de Python, algunes solucions desenvolupades (com la *bootROM* escrita en binari i injectada al simulador) poden ser millorades per funcions exposades per la part de Spike mitjançant els *bindings* que exposa PySpike.

ÚS DE LA IA GENERATIVA

Per a la realització d'aquest treball s'han fet servir eines d'IA generativa en concordança amb els criteris del tutor del TFG, específicament Claude Code per a la millora i adaptació de diversos apartats. Els casos i la metodologia emprada en el seu ús són els següents:

- Assistència en el format del disseny del document $\text{\LaTeX}$, creació de taules i correccions ortogràfiques i/o gramaticals.
- Millora i optimització dels scripts de simulació en Python.
- Assistència en l'anàlisi del funcionament de codi amb poca documentació (ex: codi font de Spike).

Cal remarcar que no s'han utilitzat eines d'IA generativa per a la planificació de la metodologia ni com a font d'informació primària o íntegrament fiable. El seu ús ha estat restringit únicament a les tasques enumerades anteriorment, amb una supervisió total per part de l'alumne sobre els resultats que genera l'eina. Sovint fa suposicions incorrectes sobre temes com el funcionament de rutines específiques de programes dels quals no revisa el codi font (com Spike), pot confondre codificacions sense tenir en compte les especificacions del sistema (arriba a confondre adreces de CSR, la qual cosa dona diagnòstics incorrectes de *bugs*) i tendeix a ser incapaç de trobar alternatives efectives a problemes quan la primera solució no funciona. Les idees de la metodologia del projecte, com l'exposició de *bindings* addicionals als implementats per PySpike per escriure els binaris directament a memòria o simular la *bootROM* per a la càrrega del *device tree*, han estat planificades íntegrament per l'alumne. Les eines d'IA generativa no tenen la capacitat de diferenciar entre el funcionament de Spike des de la línia de comandes i el Spike instrumentat que es fa servir en aquest treball, i en demanar qualsevol informació a aquesta eina també se'n demanen les fonts per comprovar la veracitat de les afirmacions rebudes pel LLM.

AGRAÏMENTS

Als meus pares, per esforçar-se en tot moment perquè avui jo pugui ser aquí. A la meua parella per donar-me suport en tot moment aquests 4 anys de carrera.

9 BIBLIOGRAFIA

REFERÈNCIES

- [1] B. Ighilahriz, O. Savry. *Comprehensive lock-step verification for NaxRiscv SoC integrating RISC-V DV, RVLS, and Questa/UVM*. RISC-V Summit Europe, May 2025, Paris, France. cea-05052859 [Online]. Available: <https://cea.hal.science/cea-05052859/document>
- [2] F. Malatesta. (2024, Sep. 17). *Accelerating RISC-V processor verification: A co-simulation strategy* [Online]. Available: <https://blog.gaisler.com/accelerating-risc-v-processor-verification-a-co-simulation-strategy/>
- [3] RISC-V International. *Spike, a RISC-V ISA simulator* [Online]. Available: <https://github.com/riscv-software-src/riscv-isa-sim>
- [4] RISC-V International. *riscv-tests: Unit tests for RISC-V processors* [Online]. Available: <https://github.com/riscv-software-src/riscv-tests>
- [5] RISC-V International. (2022). *Using spike in a lock-step hardware simulation*, GitHub Issue #911 [Online]. Available: <https://github.com/riscv-software-src/riscv-isa-sim/issues/911>
- [6] L. Yu. (2025). *pyspike: Python bindings for the Spike RISC-V ISA simulator* [Online]. Available: <https://github.com/liuyu81/pyspike>
- [7] Buildroot Project. *Buildroot: Making embedded Linux easy* [Online]. Available: <https://buildroot.org>
- [8] RISC-V International. (2025). *OpenSBI: RISC-V open source supervisor binary interface* [Online]. Available: <https://github.com/riscv-software-src/opensbi>
- [9] D. Castells-Rufas, D. Novo, and X. Martorell. *Punxa :: A Python-Based RISC-V System Simulator for Education*. Electronics Letters, vol. 61, no. 1, 2025, doi: 10.1049/ell2.70300.
- [10] D. Castells-Rufas, G. Rotger, and D. Novo. *Streamlining FPGA Circuit Design and Verification with Python and py4hw*. Proc. XI Southern Programmable Logic Conference, San Luis, Argentina, Mar. 2023.
- [11] RISC-V International. *The RISC-V Instruction Set Manual Volume II: Privileged ISA* [Online]. Available: https://docs.riscv.org/reference/isa/_attachments/riscv-privileged.pdf
- [12] SiFive, Inc. (2019). *SiFive Interrupt Cookbook* (Version 1.2) [Online]. Available via Internet Archive: <https://web.archive.org/web/20230206132717/https://starfivetech.com/uploads/sifive-interrupt-cookbook-v1p2.pdf>
- [13] Rivos Inc. (2022). *Hammer: Infrastructure to drive Spike (RISC-V ISA Simulator) in cosim mode* [Online]. Available: <https://github.com/rivosinc/hammer>
- [14] M. Oduardo. (2026). *Lockspike: Lock-step execution Python framework for Spike, the RISC-V ISS* [Online]. Available: <https://github.com/marcoos204/lockspike>