Implementació d'un Sistema RISC-V en FPGA amb suport per Linux

Oscar Lostes Cazorla

Resum– RISC-V és un ecosistema de processadors de codi obert que s'està obrint pas en els últims anys. Si bé és prou popular en àmbits acadèmics, encara no ha guanyat adeptes entre els consumidors finals. En aquest treball es planteja implementar un sistema RISC-V, aprofitant plaques FPGA disponibles, i amb capacitat per funcionar amb Linux, per tal d'aconseguir una experiència d'usuari més amigable que un entorn baremetal. S'exploren les alternatives HW i SW, així com la generació i síntesi del sistema RocketChip en una placa Altera, mitjançant el framework Chipyard. També s'analitza el procés de boot de Linux en un sistema d'aquestes característiques, si bé no s'ha pogut implementar. Finalment, s'ha assolit un prototipus capaç d'executar software baremetal i que pot interactuar de forma efectiva amb hardware extern.
Paraules clau– RISC-V, Linux, FPGA, Chipyard, RocketChip, Altera, Quartus

Abstract– RISC-V is an open-source processor ecosystem that has been making inroads in recent years. While it is quite popular in the academical environment, it has not yet gained followers among end consumers. The aim of this project is to implement a Linux-capable RISC-V system, taking advantage of available FPGA boards, in order to achieve a more user-friendly experience than a baremetal environment. The HW and SW alternatives are explored, as well as the generation and synthesis of the RocketChip system on an Altera board, using the Chipyard framework. Linux boot process on such a system is analized too, although it has not been implemented. At the end, the project achieved a prototype capable of running baremetal software and able to interact with external hardware.

Keywords– RISC-V, Linux, FPGA, Chipyard, RocketChip, Altera, Quartus

1 Introducció - Context del treball

RISC-V és un conjunt d'estàndards d'arquitectura de processadors, de naturalesa oberta i lliure, en contraposició amb arquitectures privatives com poden ser x86 o ARM.

Al llarg dels anys, s'han desenvolupat diverses implementacions de RISC-V, principalment en entorns universitaris (com la universitat ETH de Zürich [1] o Berkeley [2]), tot i que també hi ha iniciatives comercials (com SiFive [3] o Andes [4]).

Si bé l'ecosistema RISC-V és robust i prou madur, alhora presenta un problema d'usabilitat important si pretén guanyar adeptes més enllà de les esferes acadèmiques (on actualment l'arquitectura ja va tenint un pes important): els sistemes operatius.

Actualment, la majoria de sistemes implementats amb processadors RISC-V cauen en tres categories principals:

- Sistemes baremetal: dispositius corrent software directament sobre el processador, sense sistema operatiu, i que necessiten compilació de forma creuada en una plataforma tradicional.

  Aquests acostumen a ser els usos més acadèmics de RISC-V, de cara a demostrar el seu funcionament. Però aquesta forma d'ús és poc amigable per a l'usuari.

- Sistemes encastats: dispositius amb SO de baix impacte, i normalment en temps real (estil Zephyr [5], FreeRTOS [6] o NucleusRTOS [7]), orientats a aplicacions encastades.

  Si bé aquesta configuració és molt més versàtil que la baremetal, els SO encastats són bastant limitats i orientats a usos més industrials.

• E-mail de contacte: oscar.lostes@autonoma.cat
• Menció realitzada: Enginyeria de Computadors
• Treball tutoritzat per: David Castells Rufas (Departament de Microelectrònica i Sistemes Electrònics)
• Curs 2021/22

- Sistemes de propòsit general: dispositius amb SO de propòsit general (General Purpose OS, GPOS), com per exemple Linux, que s'apropen a l'experiència tradicional d'usuari.

  Aquests GPOS requereixen d'una sèrie de capacitats tècniques del dispositiu més avançades que les que pugui necessitar un SO encastat.

Si l'objectiu és apropar RISC-V a l'usuari final, la millor opció és utilitzar un GPOS. Al mercat s'hi poden trobar alguns productes que implementen hardcores RISC-V (es a dir, es fabrica el processador en silici) [8], i amb capacitat d'executar Linux de forma relativament completa i fàcil. Ara bé, aquests productes tendeixen a ser costosos i, per tant, s'allunyen del propòsit del treball.

Partint dels punts anteriors, es deriva l'objectiu principal del treball: implementar un sistema RISC-V sobre una FPGA, i donar-li una capacitat real mitjançant l'execució d'un sistema operatiu GNU/Linux.

Així mateix, la motivació principal del treball neix també dels següents factors:

- Accessibilitat: les FPGA són una de les millors alternatives quan es vol provar hardware, obviant la simulació. Són plataformes molt més barates que la fabricació tradicional, i molt més simples que implementar circuits amb components discrets i/o circuits integrats.

  Això facilita tant el desenvolupament del projecte, com el posterior ús que en pugui fer el públic general.

- Disponibilitat de recursos: el departament disposa de gran quantitat de plaques de prototipatge basades en FPGA i, per tant, el cost material que suposa aquest projecte queda cobert.

  A més, potencialment el departament pot fer ús dels resultats obtinguts de forma fàcil, ja que s'està emprant el seu material.

- Utilitat docent: més enllà d'intentar assolir un producte que pugui ser útil per al públic general, o assolir una fita tècnica interessant, aquest projecte pot tenir una gran utilitat a les aules.

  Mitjançant el coneixement adquirit en aquest treball, es podria generar material docent per ensenyar nombroses temàtiques transversals a l'ensenyament tant de l'enginyeria informàtica com a enginyeria electrònica de telecomunicacions: arquitectures de computadors, sistemes operatius, programació de baix nivell, llenguatges de descripció de hardware, ús de FPGA, disseny RTL, etc.

- Motivacions personals: aquest projecte mostra una combinació de dues de les tecnologies que més m'han fascinat al llarg del grau: FPGA i Linux. Ambdós temes es tracten durant els estudis, però no amb el detall que seria desitjable.

1.1 Objectius concrets

Donat que la premissa del treball pot ser bastant oberta, s'ha de concretar què s'espera del producte final. Aquesta descripció ve limitada per factors com poden ser material, coneixement o temps.

- L'objectiu del treball no és construir cap dels components individuals que conformen el sistema descrit, sinó integrar-los per a complir l'objectiu desitjat. Això significa que aquest treball no consisteix a dissenyar un processador RISC-V o un SO Linux.

- No s'espera que el sistema tingui gran rendiment, ni baix impacte energètic o de recursos, només que sigui funcional.

  Els softcores (processador dissenyat sense el propòsit de ser fabricat) són molt útils pel fet que es puguin sintetitzar i prototipar ràpidament, però no són coneguts per ser implementacions òptimes.

- El sistema quedarà implementat en una placa de prototipatge Altera/Terasic, amb FPGA Cyclone d'Intel, ja que és el tipus de dispositiu més abundant al departament.

- Idealment, el sistema hauria de ser independent de dispositius externs, com a mínim en termes d'arrencar el SO. El mètode objectiu és carregar el sistema des de la SDCard que incorporen la majoria de plaques FPGA.

- Per tal d'acostar-se el màxim possible a l'usuari, s'aspira a poder executar un gestor de finestres i entorn d'escriptori, i poder utilitzar-lo directament a través de teclat/ratolí i sortida de vídeo disponible a la FPGA emprada.

  Si no es pogués assolir, també serien acceptables connexions a terminal mitjançant el port sèrie.

D'aquestes característiques, se'n deriven les següents tasques:

- Determinar les característiques mínimes d'un sistema RISC-V que permetin executar Linux.

- Escollir una distribució de Linux compatible amb RISC-V.

- Escollir un softcore RISC-V obert i que complexi els requeriments anteriors.

- Aconseguir sintetitzar el processador escollit en alguna de les FPGA disponibles.

  Aquest objectiu, indirectament, implica la necessitat d'escollir una placa de prototipatge. L'elecció s'ha de fer, en part, de forma empírica, ja que és difícil saber amb anterioritat les necessitats en recursos del disseny escollit.

  Si no s'aconsegueix aquesta fita, s'haurà de revisar l'elecció del softcore.

- Implementar un sistema mínim al voltant del core que permeti l'execució de software baremetal, per tal de validar el correcte funcionament del dispositiu.

X Ampliar el sistema amb els dispositius necessaris per a carregar Linux de forma autònoma i monitorar aquesta càrrega (és a dir, un controlador SPI per a llegir de la SD de la placa, i algun mòdul de comunicació com pot ser UART o JTAG).

X Compilar la distribució escollida i comprovar que el procés de boot és satisfactori (encara que el sistema no estigui complet)

X Ampliar el sistema amb un controlador de vídeo i amb suport per a perifèrics bàsics (com a mínim un teclat), per tal de poder tenir l'experiència completa.

Al llarg del treball s'aniran desenvolupant aquests objectius, i el procés i evolució que ha anat comportant cada etapa del projecte.

Malauradament, no s'han pogut acomplir totes aquestes fites per diversos factors aliens que han alterat la normal execució del projecte. Malgrat tot, es considera que s'han obtingut resultats útils, i que tots els requisits assolits ho fan de forma satisfactòria.

2 Estat de l'art

Si bé l'esfera de RISC-V s'està expandint en els últims anys, encara està lluny d'arribar a un públic més general. Com s'ha explicat a la introducció, aquesta arquitectura queda principalment reduïda a un àmbit acadèmic, i les alternatives comercials són encara desconegudes, costoses i molt limitades.

Comparativament amb aquest treball, podem trobar projectes oberts amb aspiracions similars d'implementar softcores RISC-V en FPGA. Ara bé, la majoria cauen en una o més de les següents categories:

- Implementen processadors de 32 bits, fet que limita el seu ús per a computació de propòsit general, que actualment ja s'ha consolidat als 64 bits.

  Per exemple, VexRiscv [9] o PicoRV32 [10].

- Utilitzen processadors que no implementen les característiques requerides per poder suportar Linux.

  Per exemple, Lizard Core [11] és un RISC-V de 64 bits, però que només implementa característiques bàsiques (enters i multiplicació/divisió).

- Es sintetitzen, principalment, en dispositius FPGA de Xilinx, havent molt pocs projectes que ho facin en plaques Altera/Terasic (NEORV32 [12] és un dels pocs).

  Per exemple, el propi RocketChip que s'acabarà emprant en aquest projecte s'implementa típicament en plaques Xilinx [13].

Les dues primeres característiques són obstacles importants de cara a complir amb les motivacions anteriorment esmentades, i, per tant, aquest treball aspira a superar-les. Per altra banda, el fet de ser un dels pocs projectes (oberts) que empren dispositius Altera, el pot convertir en un referent per a futurs desenvolupaments més avançats.

3 Metodologia

Primerament, s'ha dut a terme una important tasca d'investigació, per tal de determinar les millors opcions per arribar als objectius anteriorment esmenats. S'han consultat nombroses fonts documentals (totes en línia), de caràcter tant tècnic com divulgatiu, com poden ser fulls de referència oficials de productes i software, fòrums i comunitats tècniques, presentacions, blogs de tecnologia o vídeos. Així i tot, donada la naturalesa experimental del projecte, aquesta fase és contínua i essencial durant el transcurs del projecte. Constantment s'ha de consultar la documentació tècnica per tal d'implementar alguna característica, o bé s'ha de replantejar alguna de les opcions escollides originalment i es necessita nova informació.

Paral·lelament, l'execució del treball s'ha desenvolupat mitjançant una metodologia incremental. S'han realitzat múltiples iteracions dedicades a aconseguir petites fites, i no s'avança a la següent fins que no s'acaba l'anterior. Aquestes fites, enteses com a passes necessàries per a arribar als objectius, es determinen de forma dinàmica en funció de les mateixes necessitats de l'evolució del projecte. Es plantegen així en contraposició als objectius principals del projecte, que han de quedar ben definits des del principi per tal de garantir una bona execució.

Aquest treball incremental ha permès veure una evolució consistent al llarg del treball, i sobretot, ha facilitat el descobriment i reparació d'errors. Partint de la base sòlida de les tasques anteriors que, idealment haurien de ser correctes i funcionals, les falles del sistema es troben ben localitzades. Si s'hagués optat per intentar implementar tot el projecte d'un sol cop, donada la seva magnitud, hauria estat molt més complicat.

4 Investigació de Requeriments

Abans de procedir a dur a terme cap tasca efectiva, s'ha d'obtenir informació sobre les tecnologies més adients per a dur a terme el projecte. Aquesta investigació prèvia ha permès (com es veurà més endavant) redirigir el curs d'acció de forma àgil quan han aparegut problemes, ja que en tot moment hi ha hagut consciència de les possibles alternatives.

4.1 Kernel de Linux sobre RISC-V, i distribucions compatibles

Primerament, s'ha de determinar quins requisits són necessaris per a poder executar Linux sobre un

RISC-V.

En general, Linux (i el món de la computació d'usuari en general, tal com s'ha mencionat anteriorment) ja té bastant abandonat els 32 bits. Per tant, si es pretén ser accessible al públic general, s'hauria de tendir a escollir processadors i distribucions de 64 bits.

El següent pas és identificar quines distribucions (orientades a usuari, i no d'ús industrial) són compatibles amb RISC-V de 64 bits. Les principals alternatives a tenir en compte són: Debian [14], Ubuntu [15], Fedora [16] i Gentoo [17].

D'aquestes quatre distribucions, les que estan més al dia són Debian i Ubuntu, essent la segona la més coneguda per als usuaris. Per tant, seguint l'argument principal de la usabilitat, sembla coherent emprar Ubuntu com distribució objectiu. Si això no fos possible, la següent opció acceptable seria Debian (per proximitat tècnica). Si no fos possible, sempre queden sobre la taula la resta de distribucions o, fins i tot, muntar un sistema més mínim amb projectes de l'estil BusyBox per tal de tenir un Proof-of-Concept funcional.

D'altra banda, Linux requereix una sèrie de capacitats mínimes del processador. A l'especificació RISC-V, aquestes capacitats es materialitzen en diferents extensions de l'arquitectura base (l'arquitectura base només suporta manipulació d'enters). En aquest cas, Linux requereix l'extensió de propòsit general (extensió G), que comprèn un conjunt d'altres extensions (multiplicació/divisió d'enters, aritmètica de punt flotant de simple i doble precisió, i operacions atòmiques, principalment). També se suporta, en alguns casos, l'extensió per instruccions comprimides (extensió C).

En conclusió, i seguint la nomenclatura de RISC-V, això implica que un processador compatible amb Linux (dins dels paràmetres ja establerts) ha de ser necessàriament un processador RV64GC.

4.2 Softcores amb suport Linux i FPGA on s'implementen habitualment

Analitzant els requisits de Linux anteriorment esmentats, a la taula 1 es presenten els principals softcores RV64GC (i els entorns System on Chip, SoC, al seu voltant) que s'han considerat més adients per la tasca.

Tal com s'ha comentat anteriorment, podem veure una marcada tendència cap a les FPGA de Xilinx. Per tant, si els altres mètodes no funcionen, potser s'ha d'investigar les plaques Xilinx concretes, trobar una Altera similar i intentar convertir projectes d'una placa a l'altra manualment. Ara bé, sembla que els mitjans de síntesi mostrats haurien de servir. Tant Chipyard [18] com RocketChip [19] generen codi Verilog independent de la FPGA de destí, i la llibreria GRLIB de Gaisler [20] suporta diverses plaques Altera/Terasic.

La llibreria GRLIB de Gaisler, disposa de suport explícit per a plaques Altera/Terasic, i, per tant, sembla l'opció més adequada en primera instància. En retrospectiva, i tal com es podrà veure en apartats posteriors, aquesta tecnologia no és útil per al projecte i s'ha acabat descartant.

La següent opció que sembla viable, i que ha acabat essent la utilitzada, és la plataforma Chipyard, que permet generar un SoC amb qualsevol dels altres 3 softcores. En una consulta preliminar a la documentació de Chipyard, es determina que el suport per a CVA6 és un dels més limitats.

D'entre les altres dues opcions, Rocket i BOOM presenten la mateixa interfície externa. La diferència fonamental és que BOOM és un processador fora d'ordre (Berkeley Out-Of-Order Machine) i Rocket no. Per tant, el Rocket Core sembla al més adient, ja que BOOM pot ser potencialment més complex i requerir més recursos (factor crític quan es tracta de FPGA).

5 Softcores i els seus entorns

5.1 GRLIB i NOEL-V

La GRLIB és una llibreria d'elements IP (Intellectual Property) propietat de l'empresa Gaisler. Aquesta conté una selecció de softcores dissenyats per la companyia (essent el NOEL-V [21] el model basat en RISC-V), així com components accessoris necessaris per a la construcció d'un SoC. Tal com s'ha comentat anteriorment, la llibreria presenta cert suport per a plaques Altera i pel seu software de disseny Quartus. La llibreria, pel fet de ser una iniciativa comercial, no està totalment oberta i hi ha recursos que s'han de comprar. Això no és problema, ja que la part gratuïta cobreix els elements teòricament necessaris per al projecte.

Si bé inicialment es va establir que NOEL-V semblava la millor opció, i s'hi van dedicar mesos d'esforç, s'ha acabat desistint d'emprar aquesta tecnologia pels següents motius:

- Si bé la GRLIB presenta suport per a plaques Altera, només ofereix projectes pregenerats per al LEON3 (un altre processador de la companyia). Això implica que s'ha de generar tot de zero, i si bé no és inherentment un problema, en conjunt amb la resta de motius sí que ho és.
- NOEL-V, i el SoC que s'implementa al seu voltant, és massa gran (en recursos de la FPGA) per a qualsevol de les plaques de què es disposen actualment per al projecte.
- La GRLIB proporciona una interfície de configuració (tan gràfica, com es pot veure a la figura 1, com textual) bastant limitada. Alguns dels components que es voldrien configurar per a reduir l'ús

TAULA 1: Softcores amb capacitats per executar Linux

Softcore	Developer	HW "Oficial"	Mitjà de síntesi
Rocket	UC Berkeley	Artix-7 35T Arty FPGA Xilinx Virtex-7 FPGA VC707 Xilinx Virtex UltraScale+ FPGA VCU118	Rocket Chip Generator Chipyard
BOOM			
Ariane / CVA6	ETH-Zurich	Digilent Genesys 2 Xilinx Vertex 7 – VC707	Projecte de Vivado Chipyard
NOEL-V	Gaisler	Digilent Arty-A7 Microsemi PolarFire Xilinx KCU105	Llibreria d'IP GRLIB

de recursos es troben hardcoded al disseny i no es poden canviar fàcilment.

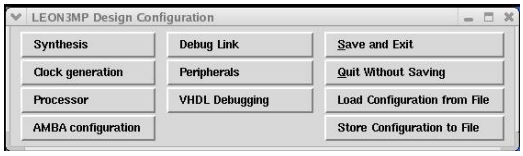

Fig. 1: Interfície XConfig de la GRLIB

- Quartus treballa, principalment, amb el bus de comunicacions AXI, mentre que el NOEL-V empra el bus AMBA.

A més, el codi VHDL de la llibreria no permet connectar de forma fàcil dispositius externs a la FPGA (com poden ser la RAM o la SD, en aquest cas). Per superar aquest problema, es va decidir generar un component per al dissenyador de plataformes de Quartus (o component Qsys). Però a l'hora de dur a terme aquesta tasca es presenta un problema de dependències amb un cert component de la llibreria, que no s'ha pogut trobar enlloc (ni als arxius, ni a la documentació, ni a Internet) i fa impossible la generació d'aquest component.

Per tots aquests motius, es va desestimar NOEL-V com a softcore candidat, i es va procedir a investigar les alternatives.

5.2 Chipyard i RocketChip Generator

Chipyard és un framework de síntesi i simulació de sistemes RISC-V, que aglutina diverses eines i llibreries en un sol paquet, desenvolupat pel Berkeley Architecture Research Group (UCB-BAR) del departament d'Informàtica i Enginyeria Elèctrica de la Universitat de Berkeley, Califòrnia.

Aquest projecte està basat en Chisel: un llenguatge de descripció de hardware implementat en Scala (que al seu torn és un llenguatge de propòsit general que corre a la Java VM). D'aquesta forma, no existeix un codi Verilog directament, sinó que aquest codi es genera d'acord amb les especificacions donades.

Mitjançant Chipyard és possible, a partir de components i configuracions descrites en Chisel, obtenir un SoC complet. En aquest SoC podem trobar els diferents softcores esmentats anteriorment, així com diversos acceleradors i un assortiment dels dispositius més elementals que puguem esperar en un sistema d'aquestes característiques (UART, SPI, JTAG, GPIO, etc.). A més, permet exposar els busos interns amb especificacions ben conegudes (com pot ser AXI). La configuració té un enfocament modular, on podem afegir, configurar i retirar elements a voluntat. Així, el framework permet muntar infinitat de dissenys: diverses arquitectures de memòria, DMA, configuracions de busos, sistemes multinucli tant homogenis com heterogenis...

El generador Rocket Chip (el SoC basat en Rocket Core) està inclòs a Chipyard. És a dir, Rocket Core no existeix com una entitat separada, sinó que s'entén com el SoC complet. Així mateix, RocketChip Generator està, com la resta de Chipyard, dissenyat en Chisel (tant Rocket com Chipyard, com el mateix llenguatge, estan elaborats pel UCB-BAR). Això implica que no hi ha una descripció de hardware prefixada del RocketChip, tal com succeeix amb NOEL-V, sinó que es genera ad-hoc per al sistema configurat.

Amb tot això, Chipyard sembla l'eina ideal per dur a terme aquest projecte. En els pròxims apartats, durant l'execució del projecte, s'anirà explorant aquest framework de forma més exhaustiva.

6 Realització del projecte

S'aniran desglossant un a un els diferents objectius del projecte, i com s'han anat assolint.

6.1 Generació i síntesi del Softcore

Tal com s'ha comentat anteriorment, les primeres proves es van realitzar amb el NOEL-V. Donat que finalment no compleix amb els requeriments d'espai de les FPGA disponibles, no s'entrarà en detall per poder explicar el funcionament de Chipyard amb marge.

Primerament, i per tal de veure com funciona el sistema, es genera el SoC d'exemple per defecte amb Rocket. Aquest proporciona tots els dispositius bàsics que es puguin necessitar, i també ens proporciona un factor clau: interconnexió amb el bus AXI, tant per a memòria principal com per a perifèrics. Això significa que, de cara a muntar un sistema sencer amb RAM, memòria secundària (SD), sortida de vídeo i

teclat/ratolí, es pot generar un component Qsys amb aquest SoC i integrar-lo fàcilment amb tots aquests dispositius.

Abans, però, de procedir a realitzar aquest component, s'ha de determinar si es pot sintetitzar aquest sistema en alguna de les plaques.

A partir de la generació per defecte del RockeChip, s'observa un ús desmesurat de recursos LE o Logic Elements (de l'ordre del 300% a les plaques amb més capacitat). Una anàlisi de l'ús individualitzat dels recursos per cada component del disseny revela que els components amb més contribució eren les memòries cache (tant de primer com de segon nivell). Aquesta observació es confirma experimentalment amb l'eliminació de la cache L2 i la reducció al mínim de la cache L1 (ja que Chipyard no permet eliminar fàcilment l'L1). Aquests canvis de configuració porten el disseny a unes dimensions acceptables.

Tot i això, aquesta solució dista de ser òptima i no explica per què les memòries disparen tant les dimensions del disseny. Després d'analitzar el codi Verilog generat per Chipyard, així com consultant la documentació, s'arriba a la conclusió que les memòries del disseny no s'estan reconeixent com a blocs de RAM: Quartus està intentant sintetitzar aquests mòduls Verilog a partir d'una descripció comportamental, que no permet que l'eina infereixi que es tracten de memòries, i, per tant, aquestes s'acaben sintetitzant com a registres. Això es tradueix en dispositius cache que utilitzen gran quantitat de recursos, ignorant els blocs de memòria disponibles a la FPGA.

Preguntant a la mailing list del projecte [22], es conclou que el mètode de generació emprat no és l'adeqüat pel cas d'ús, i es troba l'alternativa correcta. A l'apèndix A.1 es troba una explicació detallada de com s'ha realitzat aquest procés correcte de generació. També s'hi pot trobar una referència a la configuració final del SoC.

Amb aquest workflow de generació, la resta de components es mantenen pràcticament inalterats, mentre que les memòries es generen amb descripcions de flip-flops que Quartus pot inferir com a memòries correctament. Això resulta en un disseny amb dimensions acceptables, fins i tot després d'haver reactivat l'L2 i retornat l'L1 a quantitats raonables, ja que ara les caches fan servir recursos de memòria en comptes de LEs.

TAULA 2: Ús absolut de recursos

SoC	ALMs	Memory	Registers	DSPs
NOEL-V	115.167	8.608.896	152.731	10
Rocket v0	217.093	9.247	295.931	13
Rocket v1	28.004	9.208	26.694	13
Rocket vFinal	24.271	834.530	24.815	13

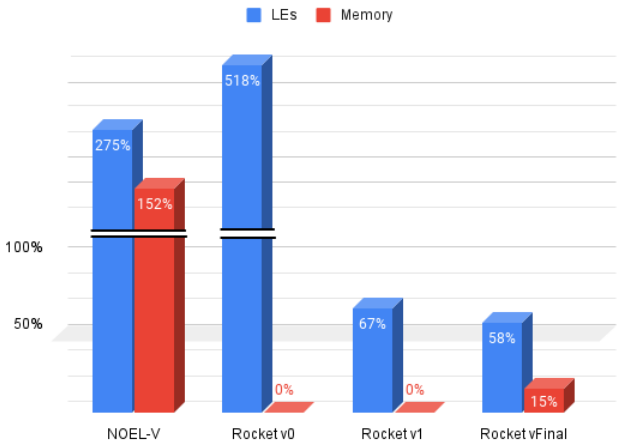

Fig. 2: Comparativa d'ús de recursos i memòria

A la figura 2 i a la taula 2 podem trobar una comparativa entre els 4 sistemes exposats:

- NOEL-V
  - Dues Cache (una de dades, i una altra d'instruccions) L1 associatives d'16 KiB i 4 vies.
  - Sense Cache L2.
- Rocket v0 (Sistema d'exemple)
  - Dues Cache L1 associatives d'16 KiB i 4 vies.
  - Cache L2 associativa de 512 KiB i 8 vies.
- Rocket v1 (Cache al mínim)
  - Dues Cache L1 associatives de 128B i 2 vies.
  - Sense Cache L2.
- Rocket vFinal
  - Igual que la v0, però forçant l'ús dels blocs RAM de la FPGA per implementar les cache.
  - Inclou una RAM On-Chip de 64 KiB que s'explicarà més endavant.

Amb aquestes dimensions, es determina que qualsevol de les plaques disponibles són candidates vàlides a allotjar el sistema. Basant-se en això, es decideix fer servir el model DE10-Nano [23], perquè és el que disposa de la millor RAM externa (SDRAM 1GB DDR3), essent la resta de característiques molt similars entre elles.

6.2 Muntatge del sistema inicial

Un cop es domina la generació del SoC, i podem garantir que la síntesi és viable, es procedeix a produir un component Qsys a partir del resultat obtingut de Chipyard. El SoC que es generarà aquest cop no és el d'exemple, sinó un configurat específicament per al cas d'ús del projecte en el punt en què es troba.

Les característiques més importants de la configuració emprada per aquest pas són:

- 1 RocketCore gran (versió amb totes les capacitats) de 64 bits.
- Dues Cache L1 associatives d'16 KiB i 4 vies.
- Cache L2 associativa de 512 KiB i 8 vies.
- Dos ports externs de tipus AXI4: el de memòria i el de perifèrics.
  - El port de memòria es configura de 32 bits, ja que la SDRAM de la DE10-Nano té aquesta amplada. Si bé aquest fet no suposa un problema (perquè les instruccions de RISC-V són de 32 bits), requereix una configuració explícita.
  - Els perifèrics es troben mapejats en memòria i, per tant, el port es coneix com a MMIO (Memory Mapped Input Output).
- S'eliminen tota la resta de busos i dispositius que conformen el SoC d'exemple, per tal de quedar-se amb una versió mínima que contingui únicament els elements necessaris.

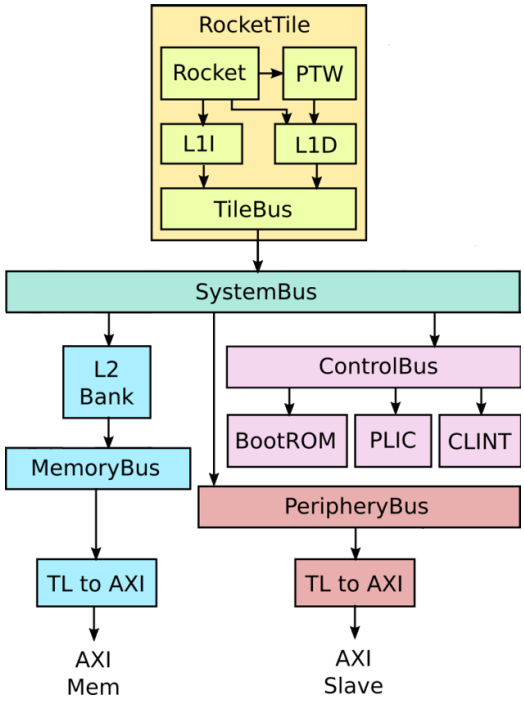

Fig. 3: RocketChip amb busos de Memòria i MMIO

A la figura 3 es pot veure una descripció d'alt nivell del sistema generat.

Importar el codi Verilog al dissenyador de plataformes de Quartus és un procés bastant directe: indicar els arxius Verilog involucrats en el disseny, seleccionar la top-level entity (ChipTop, en aquest cas), definir les interfícies (els dos busos AXI, rellotge i reset), i obtenir el component. Després, aquest component es pot portar a Quartus sense gaire dificultat i obtenir una síntesi correcta.

A partir d'aquí, es dissenya un sistema Qsys mínim al voltant d'aquest component, per tal de comprovar el funcionament. Aquest consta dels següents elements:

- El component amb el SoC (batejat com a UABC-hip).
- Una connexió amb els LED integrats a la placa de prototipatge, per tal de poder interactuar amb ells de cara a fer proves. Com disposem de 8 LED, amb 1 sol byte podem adreçar-los tots, i cada bit es correspon amb un LED. Per tant, és molt directe escriure un byte concret i veure la seva representació binària directament als LED.
- Una RAM externa (de la que en parlarem més endavant, ja que la seva implementació no ha estat trivial).
- Una PLL que generi una freqüència de rellotge adequada per aquesta RAM.
- Una RAM On-Chip (és a dir, implementada amb elements de memòria de la FPGA), que contindrà software necessari per fer proves, i arrencar el dispositiu en un futur. Aquesta RAM s'ha especificat que s'inicialitzi mitjançant un fitxer de contingut, per tal de poder programar-la a voluntat.

  Cal remarcar la necessitat d'aquesta memòria. Tal com es veu a la figura 3, el SoC generat ja disposa d'una memòria de boot. Per tant, podria semblar innecessari afegir una memòria addicional, però hi ha dos factors molt importants que avalen aquesta decisió:
  - Aquesta BootROM és de baixa capacitat (uns 10 kB), i potencialment pot ser massa petita per encabir-hi un bootloader de Linux.
  - La BootROM es troba dins del SoC i, per tant, cada cop que es vulgui canviar el seu contingut requerirà una regeneració del sistema. Amb l'ús de la memòria externa, es pot canviar el seu contingut fins i tot amb el sistema en marxa (mitjançant eines de debug de Quartus).

A la figura 4 es poden veure les connexions, així com el mapa d'adreces. Aquestes adreces s'assignen dins dels rangs que espera el bus on es connecta el dispositiu: de 0x60000000 a 0x7fffffff per al bus MMIO, i de 0x80000000 en endavant per al bus de memòria.

6.3 Connexió amb la SDRAM

Un cop es té aquest sistema, la intenció és executar software que faci encendre els LED, per verificar que tot funciona. Ara bé, resulta que la RAM externa, tot i que sembla que ja està correctament connectada, encara no és accessible des de la FPGA.

La DE10-Nano disposa d'un processador ARM integrat i una sèrie de recursos addicionals (el que es coneix com a Hard Processor System, HPS) que no

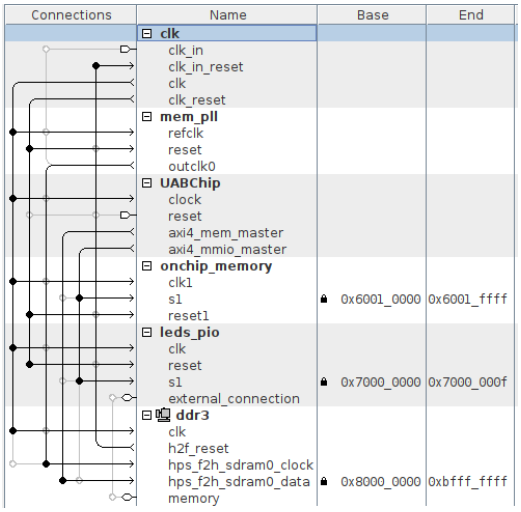

Fig. 4: Qsys del sistema mínim

són directament accessibles des de la FPGA, entre ells la RAM de la placa. És per aquest motiu que a la figura 4, al component ddr3, es fa referència al terme HPS. Aquest component prové dels exemples oficials [24], i implementa un subsistema Qsys que cedeix la RAM a la FPGA.

Ara bé, l'HPS conté el controlador de memòria amb el qual s'accedeix a la RAM i, per tant, se li han de fer certes configuracions a l'hora d'arrencar la placa, per tal de deslligar totalment els dos dispositius. Això requereix proporcionar un script de boot per al processador ARM que modifica uns certs registres per completar la transferència de control de la RAM, i fer-la totalment accessible des de la FPGA. Aquest script també es troba subministrat en el mateix exemple emprat per al bloc Qsys. s'ha de ficar a la SD de la placa, de tal forma que el processador el trobi a l'hora d'arrencar-la.

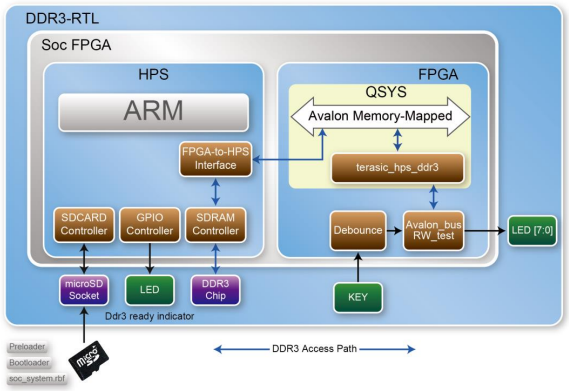

Fig. 5: Arquitectura de l'exemple de cessió de la RAM

A la figura 5 es pot veure l'arquitectura que relaciona l'HPS amb la FPGA a l'exemple emprat com a base. Si bé el sistema no és exactament el mateix, aquest diagrama permet fer-se una millor idea del funcionament d'aquesta estructura.

De la mateixa manera que es pot proporcionar aquest script, val la pena remarcar que també es pot proporcionar la configuració de la FPGA (en format .rbf), i així no haver de dependre de l'ordinador per programar el sistema. Com en aquest punt del projecte es necessita accés a les eines de depuració de Quartus, aquest mètode encara no s'utilitza.

6.4 Proves de software

Amb el sistema mínim funcionant, es procedeix a fer el test amb software. Es realitzen diverses proves, per a verificar de forma incremental diferents components i connexions del sistema. El procés d'execució ideal (amb la configuració actual) és el següent:

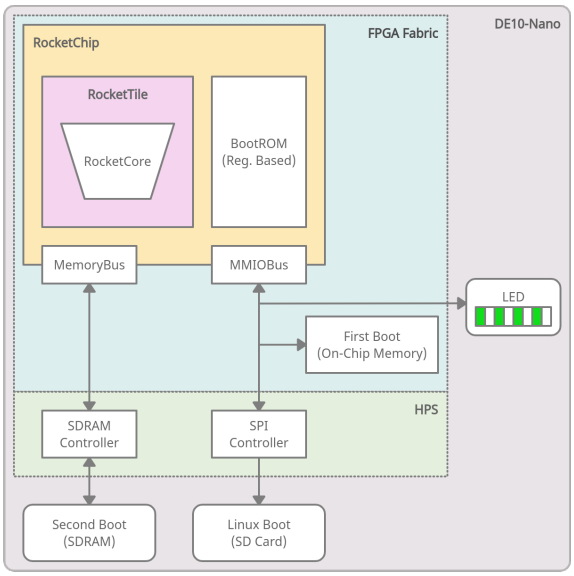

Fig. 6: Arquitectura del sistema

1. Arrencar el sistema, saltant, per defecte, l'execució a la BootROM.
2. La BootROM només s'encarrega de saltar a la RAM On-Chip.
3. La RAM On-Chip conté un programa que s'encarrega d'escriure el programa final a la SDRAM.

   A les proves inicials, el programa es troba hard-coded a la pròpia memòria. Ara bé, a l'hora d'executar Linux serà necessari carregar el sistema des de la SDCard, i per tant es necessitarà un software més avançat.
4. Es salta a la RAM externa i es comença l'execució del software desitjat.

A la figura 6 es poden veure els components involucrats en aquest procés.

Com tots aquests passos són complicats, i propensos a presentar errors, es desenvolupa un banc de proves incrementals. Totes tenen el mateix objectiu: encendre els LED de la placa amb un patró altern (10101010).

El que canvia en cada prova és el dispositiu principalment involucrat en l'execució. A l'apèndix A.2 es troba una explicació detallada de les proves realitzades i el codi ensamblador emprat.

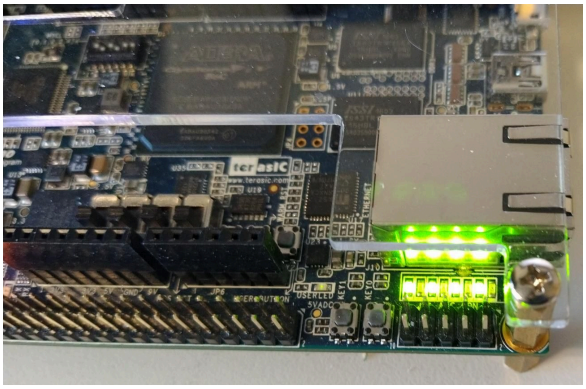

Fig. 7: Conjunt LED encès mitjançant software

Aquestes proves, tal com es pot veure a la figura 7, han resultat amb èxit i, per tant, es pot determinar que la totalitat del sistema funciona correctament.

Un punt complicat d'aquestes proves és la inicialització de la RAM On-Chip. Aquestes memòries, quan s'inicialitzen des d'un fitxer (com és el cas), utilitzen un format propietari d'Intel (Memory Initialization File, o .mif). En conseqüència, quan es compila el codi de les proves, no es pot traslladar directament el binari generat, sinó que s'ha de convertir abans en aquest format.

A això se li suma una qüestió pròpia d'ordre del RISC-V. La memòria On-Chip té paraules de 64 bits i, per tant, una paraula pot contenir fins a dues instruccions (de 32 bits). Però, s'ha de tenir en compte que el sistema és little-endian, el que implica que la primera instrucció es troba a la part baixa de la paraula, mentre que la segona es troba a l'alta. Això entra en conflicte amb la forma intuïtiva, i amb l'ordenació del binari generat.

Sabent tot això, és menester realitzar una adequació entre la compilació i el fitxer .mif. El procés complet queda descrit a l'apèndix A.2, juntament amb la resta d'informació.

7 Booting de Linux

Si bé el sistema teòricament té suport per a executar Linux, no s'ha pogut comprovar experimentalment per una manca de temps derivada de diversos factors aliens. Així i tot, s'ha investigat el procés i com s'hauria d'adaptar el sistema per a fer-ho possible (en termes generals).

Per començar, es necessita afegir dos components al sistema:

- Un SPI, per tal de poder llegir dades de la SDCard de la placa (que actuarà com a memòria secundària del sistema).
- Una UART, per tal de poder establir comunicació amb el SO que s'està carregant, des d'un dispositiu extern.

Per incloure aquests components es pot recórrer a dues alternatives: generar els dispositius afegint-los a la configuració de Chipyard, i que quedin integrats al sistema; o emprar blocs IP d'Intel, i connectar-los a través del bus MMIO del sistema.

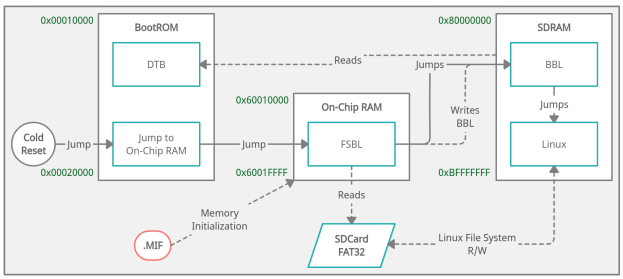

Fig. 8: Descripció gràfica del procés de boot

Amb aquests components, es pot procedir a iniciar el procés de boot. A continuació s'exposa com hauria de succeir aquest procés, basat en el procés definit a la documentació de Chipyard:

1. El processador arrenca i salta per defecte a la BootROM. Aquesta conté un salt a la RAM On-Chip. També conté una descripció dels dispositius del sistema (que es coneix com a Device Tree Binary, DTB), que més endavant permetrà a Linux saber de quins components disposa i quins drivers necessita per a fer-los funcionar correctament.
2. Mitjançant un fitxer .mif, com els utilitzats a les proves anteriors, s'inicialitza la RAM On-Chip. Aquesta conté software (conegut com a First Stage BootLoader, FSBL) dedicat a copiar dades des de la SDCard a la SDRAM. Aquest software ha de controlar l'SPI a baix nivell (ja que encara no es disposa d'abstraccions de sistema de fitxers o similars).

   El programa ha de copiar dos elements al principi de la SDRAM: el Second Stage Bootloader que carregarà Linux, i la imatge del Kernel (el que es coneix com a fitxer Vmlinux).

   Un cop fet aquest moviment de dades, es pot saltar a la SDRAM, i executar el bootloader.
3. El bootloader s'encarrega de preparar la configuració del processador necessària per a poder entrar al mode supervisor (ja que, quan el sistema arrenca, ho fa en mode màquina, sense cap protecció), així com alguns temes de memòria virtual i paginació. També s'encarrega de processar el DTB i ubicar-lo on Linux l'espera.

   Hi ha diferents opcions de bootloader ja existents, com pot ser U-Boot [25]. Ara bé, el software recomanat des de Chipyard és BBL (Berkeley Boot

Loader) [26], ja que està desenvolupat pel mateix equip que la resta del projecte.

8 Resultats i conclusions

Al final de tot el procés s'ha acabat obtenint un sistema RISC-V funcional, que és capaç de dur a terme un procés de boot simple i executar software baremetal. A més, si s'inclou la càrrega de la configuració des de la SDCard, el sistema pot ser potencialment autònom. Així mateix, ha quedat demostrat que el sistema té capacitat per treballar amb hardware extern sense problemes. Si bé no s'ha pogut testejar el sistema amb Linux a causa de complicacions, hi ha confiança en la capacitat per assolir aquest objectiu amb una mica més de feina.

Fins i tot amb els objectius que no s'han pogut assolir, es considera que el treball s'ha resolt de forma satisfactòria. El producte final il·lustra que és possible aconseguir de forma relativament simple sistemes RISC-V en FPGA, el que permet llibertat de disseny i proves. Una implementació soft-core com l'emprada en aquest treball no és de gran rendiment, però pot tenir un gran valor com a punt d'entrada a l'arquitectura, o per l'aprenentatge en entorns docents.

A continuació, es plantegen algunes línies d'actuació que podrien agafar el relleu que deixa aquest Treball de Fi de Grau:

- Finalitzar les fites que han quedat pendents. El treball està en un punt prou sòlid per a ser continuable de forma directa.
- Fer anàlisi de rendiment en comparació amb altres plataformes RISC-V, o amb altres soft-cores. Per exemple, es podria intercanviar el RocketCore per un BOOM i comparar la diferència entre un processador in-order i out-of-order.
- Desenvolupar perifèrics o acceleradors per al sistema mitjançant la FPGA, i programar els drivers per a Linux.
- Desenvolupar material docent per a l'ensenyament de temàtiques com arquitectura de computadors, sistemes operatius o integració HW/SW, així com per l'aprenentatge a baix nivell de l'arquitectura RISC-V.
- En general, aquest treball pot servir com a punt de partida per a qualsevol projecte que requereixi RISC-V, o necessiti integrar components dissenyats en RTL amb un processador.

Per acabar, es valora l'experiència de forma molt positiva. S'han reforçat conceptes i habilitats adquirits al llarg de diverses assignatures del grau, així com se n'han adquirit de nous. RISC-V encara és un món bastant desconegut, i aquest treball ha estat un molt bon inici. No es descarta continuar enfocat en aquest sector en un futur, ja que té molt potencial malgrat que quedi molt per fer.

Agraïments

Al David, que fins i tot fet pols ha estat al peu del canó, i ha portat això a bon port.
Als companys del Consell d'Estudiants, que han hagut d'aguantar massa dissertacions sobre aquest projecte.
Als meus pares, que sense entendre un borrall han estat sempre per escoltar el que hagués d'explicar.
I a Tu, que llevat la distància sempre estàs al meu costat, il·luminant els dies més grisos.

Referències

[1] https://pulp-platform.org/

[2] https://bar.eecs.berkeley.edu/projects.html

[3] https://www.sifive.com/

[4] http://www.andestech.com/en/risc-v-andes/

[5] https://risc-v-getting-started-guide.readthedocs.io/en/latest/zephyr-introduction.html

[6] https://www.freertos.org/Using-FreeRTOS-on-RISC-V.html

[7] https://https://www.plm.automation.siemens.com/global/es/products/embedded/nucleus-rtos.html

[8] https://www.sifive.com/boards/hifive-unmatched

[9] https://github.com/SpinalHDL/VexRiscv

[10] https://github.com/YosysHQ/picorv32

[11] https://github.com/cornell-brg/lizard

[12] https://umarcor.github.io/neorv32/#_fpga_implementation_results

[13] https://chipyard.readthedocs.io/en/latest/Prototyping/index.html

[14] https://wiki.debian.org/RISC-V

[15] https://wiki.ubuntu.com/RISC-V

[16] https://fedoraproject.org/wiki/Architectures/RISC-V

[17] https://wiki.gentoo.org/wiki/Project:RISC-V

[18] https://chipyard.readthedocs.io/en/latest/

[19] https://github.com/chipsalliance/rocket-chip

[20] https://www.gaisler.com/index.php/products/ipcores/soclibrary/soclibrary-overview

[21] https://www.gaisler.com/index.php/products/processors/noel-v

[22] https://groups.google.com/g/chipyard

[23] https://de10-nano.terasic.com/

[24] DE10-Nano User Manual, pàg. 68. Terasic, 2017.

[25] https://github.com/u-boot/u-boot/

[26] https://github.com/riscv-software-src/riscv-pk

Apèndix

A.1 Configuració i generació de SoC amb Chipyard

Chipyard involucra moltes eines diferents (generadors, simuladors, compiladors, etc.) i, per tant, es pot utilitzar de diverses formes. Inicialment, es va emprar el generador RocketChip directament per a obtenir un sistema d'exemple. Però, tal com s'ha discutit anteriorment, aquest workflow dona problemes amb la síntesi de memòries.

El mètode correcte pel cas d'ús del projecte és fer servir el generador Chipyard (és a dir, el framework Chipyard conté un generador anomenat Chipyard, i que serveix com a punt d'entrada per a tota la resta de generadors). Aquest generador necessita una descripció estructural del sistema que es vol aconseguir, mitjançant la combinació de diferents configuracions, amb els components que es volen i les seves característiques.

```
// Configuració del sistema
class UABConfig extends Config(

  ...

  /* Bus de memòria de 32b */
  new With4BMemPort ++
  /* Amb BootROM */
  new WithBootROM ++
  /* Jerarquía de busos estàndar */
  new WithMulticlockCoherentBusTopology ++
  /* Elimina TLB L2 */
  new WithL2TLBs(0) ++
  /* Rellotges derivats del principal */
  new WithNoSubsystemDrivenClocks ++
  /* Sense mòduls de Debug */
  new WithNoDebug ++
  /* Sense interrupcions */
  new WithNExtTopInterrupts(0) ++
  /* Sense port per a DMA */
  new WithNoSlavePort ++
  /* Config. base de RocketChip */
  new BaseConfig
)

class UABChip extends Config(
  /* RocketCore RV64GC */
  new WithNBigCores(1) ++
  /* Configuració custom */
  new UABConfig
)
```

Fig. 9: Fragment de configuració

A la figura 9 es pot veure la part important de l'arxiu de configuració. El workflow actual, a més del mateix sistema, també genera un mòdul de test, que no és necessari per a aquest projecte, però que s'ha de configurar igualment. Aquesta configuració s'ha omès, ja que no és rellevant.

Un cop construïda la configuració, s'ha de generar el codi Verilog. Contraintuïtivament, aquest procés s'ha de realitzar des de les eines de simulació, i no des de les de generació. Si bé el framework conté exemples per a FPGA, Chipyard no està orientat a l'ús amb aquests dispositius.

Llavors, la forma de procedir és emprar el workflow de simulació amb Verilator (un simulador cycle-accurate de Verilog), i detenir el procés just quan es genera el codi Verilog. Chipyard preveu aquest procediment, i el Makefile del workflow conté una opció específica (tal com es pot veure a la figura 10).

```
make verilog CONFIG=UABChip
```

Fig. 10: Comanda de generació de codi Verilog

D'aquest procés s'obtenen múltiples arxius. Molts d'aquests només són informes de generació, informació addicional del sistema generat, o altres elements accessoris, entre ells el Device Tree Source (DTS), que posteriorment acabarà esdevenint el DTB per al procés de boot de Linux. Els arxius més importants són els de codis Verilog i SystemVerilog, dels quals només són necessaris uns quants. Tal com s'ha comentat anteriorment, el sistema generat conté un mòdul de test, que no és necessari per al projecte. Per tant, només cal agafar els arxius rellevants per la part principal del sistema (que es coneix com a ChipTop) i importar-los a Quartus i a Qsys.

Per tal de facilitar tot aquest procés de generació i selecció d'arxius, s'ha creat un Makefile senzill que l'automatitza.

A.2 Banc incremental de proves de software

Aquestes proves tenen com a objectiu validar el funcionament i correcta connexió entre tots els components del sistema emprant software que encén uns LED mitjançant diferents mètodes.

Ja que aquests LED es troben mapejats en memòria, i es disposa de 8 LED, llegir i escriure un byte concret representa totalment l'estat d'aquests LED. Per tant, les proves es basen a canviar quin dispositiu escriu aquest byte, i d'on prové.

Per tal de facilitar la generació dels arxius necessaris per a les proves, els codis venen acompanyats de Makefiles, basats en aquells que veníen per defecte amb Chipyard.

A.2.1 Execució en BootROM

```
li s0, 0xAA     // 0xAA = 10101010
li s1, LED_BASE // 0x70000000
sw s0, 0(s1)    // *(0x70000000) = 0xAA
```

Fig. 11: Codi del Test 1 (BootROM)

En aquesta primera prova, el valor dels LED es troba harcoded a la ROM, i quan el processador arrenca s'escriu aquest valor a l'adreça on es troben mapejats els LED.

A.2.2 Lectura de RAM On-Chip

```
  li s0, 0xAA       // 0xAA = 10101010
  li s1, BRAM_BASE  // 0x60010000
  li s2, LED_BASE   // 0x70000000
  sw s0, 0(s1)      // *(0x60010000) = 0xAA

.LED_LOOP:
  lb s0, 0(s1)      // s0 = *(0x60010000)
  sw s0, 0(s2)      // *(0x60010000) = s0
  j .LED_LOOP
```

Fig. 12: Codi del Test 2 (BootROM)

En aquest cas, el software de la BootROM carrega el valor del LED a la RAM On-Chip. Posteriorment, es llegeix de la RAM aquest valor, i s'escriu als LED.

Pel fet de fer servir memòria de la FPGA, mitjançant les eines de debug de Quartus (In-System Memory Content Editor) es pot modificar el contingut d'aquesta RAM en temps real. Com la lectura del valor es fa en bucle, es pot modificar el valor abans de la propera lectura, canviant així els LED que s'encenen.

A.2.3 Execució en RAM On-Chip

Ara es disposa de 2 codis diferents. A la BootROM, hi ha un codi que salta incondicionalment a la RAM On-Chip, de tal forma que l'execució es reprèn al codi que conté aquesta memòria. D'aquesta forma, es valida que el processador determina les adreces d'aquesta memòria com una regió executable i que, per tant, pot contenir codi.

```
_start:
  li s1, LED_VAL_ADDR // 0x60010020
  li s2, LED_BASE     // 0x70000000

.LED_LOOP:
  lb s0, 0(s1)        // s0 = *(0x60010020)
  sb s0, 0(s2)        // *(0x70000000) = s0
  j .LED_LOOP
```

Fig. 13: Codi del Test 3 (On-Chip)

El codi de la RAM llegeix una adreça (continguda dins de la mateixa memòria) que, un cop modificada des de Quartus, contindrà el valor dels LED.

A.2.4 Lectura de SDRAM

Ara, un cop l'execució salta a la RAM On-Chip, s'intenta accedir a la SDRAM. En aquest cas no es disposa d'un accés directe al contingut de la memòria, sinó que totes les dades les ha d'inicialitzar el processador mitjançant codi.

De forma anàloga al test 2, el codi de la RAM On-Chip escriu el valor dels LED a la SDRAM. Posteriorment, llegeix de la mateixa posició de memòria per a

```
_start:
  li s1, LED_BASE // 0x70000000
  li s2, DDR_BASE // 0x80000000

  li t0, 0xAA     // 0xAA = 10101010
  sw t0, 0(s2)    // *(0x80000000) = 0xAA

.LED_LOOP:
  lw s0, 0(s2)    // s0 = *(0x80000000)
  sw s0, 0(s1)    // *(0x70000000) = s0
```

Fig. 14: Codi del Test 4 (On-Chip)

recuperar el valor i encendre els LED. Com en aquest cas no es pot modificar el valor dels LED de forma arbitrària, el registre d'escriptura i de lectura són deliberadament diferents, de tal forma que no es pugui confondre una lectura fallida amb un valor residual de l'escriptura.

A.2.5 Test complet

Aquest test consisteix a fer que la RAM On-Chip copiï el programa final a la SDRAM, saltar-hi i executar el software des d'allà. Per això, primerament s'ha d'obtenir el codi màquina del software de la SDRAM.

```
_start:
  li s1, LED_VAL_ADDR   // 0x80000030
  li s2, LED_BASE       // 0x70000000

  li t0, 0xAA           // 0xAA = 10101010
  sw t0, 0(s1)          // *(0x80000030) = 0xAA

.loop:
  lb    s0,0(s1)        // s0 = *(0x80000030)
  sb    s0,0(s2)        // *(0x70000000) = s0
  j .loop
```

Fig. 15: Codi del Test 5 (SDRAM)

El programa realitza les mateixes tasques que el test anterior, però ara treballant en l'espai d'adreces de la SDRAM. Empíricament, es determina que el codi ocuparà els primers 20 bytes de la SDRAM i, per tant, el valor dels LED s'ubiquen més enllà d'aquesta frontera. A partir d'aquest codi, s'ha d'obtenir la seva representació binària per a poder-la escriure a la memòria.

```
0000000080000000 <.text>:
   80000000:   0aa00293    li  t0,170
   80000004:   0010049b    addiw   s1,zero,1
   80000008:   01f49493    slli    s1,s1,0x1f
   8000000c:   03048493    addi    s1,s1,48
   80000010:   70000937    lui s2,0x70000
   80000014:   00548023    sb  t0,0(s1)
   80000018:   00048403    lb  s0,0(s1)
   8000001c:   00890023    sb  s0,0(s2)
   80000020:   ff9ff06f    j   0x80000018
```

Fig. 16: Compilació del codi (figura 15)

Ara, aquest binari s'ha de copiar a la SDRAM. Per a fer-ho, la RAM On-Chip conté el programa harcoded a les instruccions d'escriptura. S'entén que aquesta opció és molt rudimentària i que dificulta canviar el software

de la SDRAM. Per sort, aquest codi només s'utilitza en aquest context de proves i, per tant, compleix la seva finalitat.

```
_start:
  li s0, DDR_BASE   // 0x80000000

  li s1, 0x0AA00293 // li t0,170
  sw s1, 0x00(s0)   // 0x80000000
  li s1, 0x0010049B // addiw s1,zero,1
  sw s1, 0x04(s0)   // 0x80000004
  li s1, 0x01F49493 // slli s1,s1,0x1f
  sw s1, 0x08(s0)   // 0x80000008
  li s1, 0x03048493 // addi s1,s1,48
  sw s1, 0x0c(s0)   // 0x8000000c
  li s1, 0x70000937 // lui s2,0x70000
  sw s1, 0x10(s0)   // 0x80000010
  li s1, 0x00548023 // sb t0,0(s1)
  sw s1, 0x14(s0)   // 0x80000014
  li s1, 0x00048403 // lb s0,0(s1)
  sw s1, 0x18(s0)   // 0x80000018
  li s1, 0x00890023 // sb s0,0(s2)
  sw s1, 0x1c(s0)   // 0x8000001c
  li s1, 0xFF9FF06F // j 0x80000018
  sw s1, 0x20(s0)   // 0x80000020

  jr s0             // Salt a la SDRAM
```

Fig. 17: Codi del Test 5 (On-Chip)

A.2.6 Compilació i continguts de memòria

Chipyard conté tota la Toolchain GCC per a RISC-V, de forma que es disposa de tot el software necessari per a generar binaris RISC-V. Amb aquesta Toolchain hauria estat possible escriure els programes de test amb C, però donada la simplicitat i el baix nivell, s'ha preferit treballar directament en llenguatge ensamblador.

De cara a la BootROM, Chipyard s'encarrega de generar la ROM amb el codi indicat (és a dir, el contingut de la memòria es decideix abans de sintetitzar, i no es pot canviar). El sistema agafa l'arxiu .S necessari i l'integra al procés de generació.

Ara bé, el procés no és tan directe per al contingut de la RAM On-Chip. El format .mif d'Intel no és un estàndard i, per tant, GCC no el genera. Aquest fitxer conté un llistat ordenat del contingut de la memòria (de la mateixa manera que el codi de la figura 17).

```
DEPTH = 16;
WIDTH = 64;
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;
CONTENT BEGIN
0000: 01F4141300100418 2934849B0AA004B7 001004B700942023 0094222349B4849B;
0004: 4934849B01F494B7 030484B700942423 0094262349348498 9374849B700014B7;
0008: 005484B700942823 00942A2302348498 4034849B000484B7 008904B700942C23;
000C: 00942E2302348498 9FF4849B001004B7 06F4849300C49493 0004006702942023;
END;
```

Fig. 18: MIF del codi final (On-Chip)

Com no es genera automàticament amb les eines disponibles, s'ha de trobar alguna (o crear l'arxiu de forma manual). Per sort, s'ha trobat l'eina de codi lliure SRecord, dedicada a la manipulació d'arxius de memòria, i que suporta el format .mif.

Un altre problema amb aquesta RAM és, tal com s'ha indicat anteriorment, una qüestió d'ordenació de les dades. La memòria és de 64 bits, i les instruccions són de 32 bits. Conseqüentment, en una sola adreça hi caben dues instruccions que es poden guardar en 2 ordres diferents.

RISC-V interpreta la part baixa del contingut com la primera instrucció, i l'alta com la segona. Per tant, dins de cada adreça, s'han d'escriure les instruccions en l'ordre invers a l'habitual (entenent que, representant els valors d'esquerra a dreta, a la part alta quedaria la primera instrucció, i no la segona).

Per sort, la mateixa eina SRecord permet expressar aquest canvi d'ordre. Això permet que es puguin generar els arxius .mif automàticament sense problemes, que després es carreguen al disseny des de Quartus.

```
srec_cat input.img -binary -byte-swap 8 -o output.mif -mif 64
```

Fig. 19: Comanda de generació del MIF