Disseny i implementació d'un processador RISC-V educatiu en una FPGA

Juny 2019

Josep Sans i Prats

dirigit per
JOSEP-LLORENÇ CRUZ DÍAZ

25 de juny de 2019

Resum

Després d'una breu familiarització amb l'arquitectura RISC-V, organització, codificació d'instruccions, tractament d'immediats, etc. El que pretén aquest document és il·lustrar sobre una possible implementació de caire acadèmic de l'ISA RISC-V. Com la implementació d'aquesta nova arquitectura es desenvoluparà amb el suport d'una placa de desenvolupament que incorpora a més a més d'una FPGA, altres dispositius d'entrada i sortida, part del desenvolupament també es dedicarà a implementar i documentar els diferents controladors per interactuar amb aquests dispositius. Obtenint finalment la implementació d'un *System on Chip*. Un cop finalitzada la implementació dels diferents mòduls, es verificarà la implementació amb diferents jocs de proves.

Resumen

Después de una breve familiarización con la arquitectura RISC-V, organización, codificación de instrucciones, tratamiento de los inmediatos, etc. Lo que pretende este documento es ilustrar sobre una posible implementación de carácter académico de l'ISA RISC-V. Como la implementación de esta nueva arquitectura se desarrollará con el soporte de una Placa de desarrolló que incorpora además de una FPGA, otros dispositivos de entrada i salida, parte del desarrolló también se dedicará a implementar i documentar los diferentes controladores para interactuar con sendos dispositivos. Obteniendo finalmente la implementación de un *System on Chip*. Una vez finalizada la implementación de los diferentes módulos, se verificará la implementación con diferentes juegos de pruebas.

Abstract

After a brief familiarization with RISC-V architecture, organization, instruction codification, management of immediate, etc. This document wants to illustrate about a possible implementation of the ISA RISC-V. Because the implementation of this new architecture will be developed with the support of a development board which incorporates different input output devices apart from an FPGA, part of the development will be focused in implementing and documenting the different controllers to interact with these devices. Finally, obtaining the implementation of a System on Chip. Once the development of the different modules is finished there will be a verification of the implementation using different test cases.

Índex

Capítol 1

Introducció

En una industria on la majoria d'eines per tal de desenvolupar software, editors de codi, compiladors, sistemes operatius, etc. són gratuïts i de codi obert. Trobem a faltar que, el que potser sigui la part més important de tot l'engranatge, el processador, no hi hagi hagut cap iniciativa, amb suficient recolzament[1], per tal de instaurar una arquitectura que serveixi com a estàndard en la comunitat *open source*, així com ho és Linux amb els sistemes operatius.

1.1 Context

Aquest projecte és un TFG (*Treball Final de Grau*) desenvolupat al DAC (*Departament d'Arquitectura de Computadors*) de la FIB (*Facultat d'Informàtica de Barcelona*). La finalitat d'aquest projecte es crear un processador el qual implementi una arquitectura RISC-V[1] (*pronunciat "risc five"*) lleugerament modificada per tal de poder ser implementada en una FPGA per alumnes que estiguin cursant l'especialitat d'Enginyeria de Computadors de la FIB.

La necessitat d'un ISA (*Instruction Set Architecture*) públic i amb una llicencia no restrictiva cada cop es més alta en els àmbits acadèmics i d'investigació. Ja que encara que l'ús d'un ISA comercial existent al mercat té uns certs beneficis com ara un gran ecosistema d'eines de desenvolupament així com un gran grup d'aplicacions o l'existència de grans fonts de documentació i exemples, els desavantatges que comporta són encara més grans:

- **Els ISAs comercials són propietaris**: Això comporta que la majoria dels propietaris no permetin una implementació oberta de la seva arquitectura i per tant aquells grups que no confiïn en la implementació del propietari de l'ISA, no podran crear la seva pròpia implementació.
- **Els ISAs comercials estan presents només en uns certs mercats**: Com podem observar, no existeix cap arquitectura la qual estigui present tant en el mercat mòbil, com en els servidors. Encara que tant *Intel* com *ARM* estan intentant entrar en el

[1]Existeixen ISAs de codi obert públics, com ara *OpenRISC*. Tot i que la seva popularitat es més aviat reduïda.

mercat de l'altre, a dia d'avui el mercat mòbil està monopolitzat per arquitectures *ARM* i el mercat de servidors està monopolitzat per arquitectures derivades del *Intel x86*.

- **Els ISAs comercials, no duren per sempre**: Arquitectures les quals han perdut popularitat com *SPARC* o *MIPS*, així com d'altres que ja no hi són com *Alpha*. Al ser propietàries, s'ha perdut la major part de la seva propietat intel·lectual i ecosistema d'eines per tal de que algun tercer pugui tornar-les ha reviure i utilitzar. Tot i que un ISA obert pot perdre popularitat, sempre es podrà recuperar en un futur per tal de continuar utilitzant i desenvolupant el seu ecosistema.

- **Els ISAs comercials són complexes**: La implementació hardware d'un ISA comercial com ara *x86* o *ARM* per tal de suportar les eines software i sistemes operatius actuals és molt complicada degut a la retrocompatibilitat. A més a més aquest grau de complexitat sovint ve donat per males decisions en l'ISA i no per característiques que milloren la seva eficiència.
  L'arquitectura RISC-V, una arquitectura moderna, té en compte els coneixements adquirits en el disseny de processadors durant els anys i els errors comesos, aconseguint un ISA nou, net i simplificat.

- **Els ISAs comercials no estan pensats per tal d'afegir-hi extensions**: El fet que les primeres versions dels ISAs comercials com *x86* i *ARM* no varen ser pensades per ser ampliades en un futur, a comportat que l'introducció de noves instruccions, afegís complexitat a l'hora de codificar les instruccions.

Per tal de donar solucions a totes aquestes problemàtiques i intentar instaurar un ISA de caràcter *open source* al mercat, l'any 2010 a l'universitat de Califòrnia a Berkley naixia el projecte RISC-V, un ISA *open source*, amb el qual es pretén donar suport tant a la comunitat educativa com als investigadors.

RISC-V és una arquitectura una mica inusual a les que actualment existeixen al mercat, doncs aquestes últimes tenen un disseny incremental, on els nous processadors no només implementen les noves extensions, sinó que incorporen totes les instruccions de les versions anteriors, pel contrari, la idea de disseny del RISC-V és una arquitectura modular. El nucli principal de RISC-V, anomenat RV32I (RV64I per a la versió de 64 bits), el qual està congelat i no canviarà en el futur, proveeix de totes aquelles instruccions necessàries per tal de desenvolupar un *stack* de software complet. La modularitat de l'arquitectura ve donada a les extensions opcionals estandarditzades que poden ser incorporades a les necessitats de cada aplicació.

1.2 Objectius

L'objectiu principal d'aquest TFG consisteix en implementar un processador amb l'ISA RISC-V a una FPGA de la forma més senzilla i entenedora possible a fi de que pugui ser reproduïda per un estudiant de PEC.
De tota l'especificació del RISC-V, s'implementaran les instruccions que es contemplen al

mòdul base de 32 bits (*RV32I*), el mòdul de multiplicació i divisió (*RV32M*) i, d'haver-hi temps suficient, algunes instruccions de coma flotant del mòdul (*RV32F*). S'han triat aquests mòduls i instruccions ja que es creu que és una càrrega de treball assolible, tant en temps com en dificultat, durant el transcurs d'un quadrimestre, el temps que dura una assignatura com PEC.
Per tal de poder utilitzar els diferents dispositius que incorpora la placa de desenvolupament, una placa la qual incorpora una FPGA així com també diferents dispositius com ara memòries SDRAM, SRAM o FLASH, connectors VGA o PS2, LEDs, etc. s'hauran d'implementar també els controladors corresponents per a aquests dispositius, ja que aquests ens ajudaran a verificar la nostra implementació.

1.3 Llenguatges HDL

Davant la necessitat de realitzar una abstracció dels circuits digitals, els quals havien anat adquirint gran complexitat, a finals dels anys 60 començaren a aparèixer els primers llenguatges que permetien descriure hardware, anomenats HDL (*Hardware Description Language*). Aquests llenguatges permeten documentar les interconnexions i el comportament d'un circuit realitzant una abstracció anomenada RTL (*Register-Transfer Level*) la qual modela el circuit en termes de flux de senyals entre els registres i les operacions que s'hi apliquen. Aquesta abstracció podrà esser traduïda a nivell de portes lògiques mitjançant una eina de síntesi que arribarà a convertir-se gràcies a altres eines de *fitter* i *routing* en un circuit integrat.

Tot i que la seva aparició data de finals dels anys 60, no va ser fins al cap d'una dècada que es van començar a popularitzar, gràcies a popularització també dels PLDs (*Programmable logic devices*), els antecessors a les FPGAs. Cap a mitjanç dels anys 80, apareixia el primer HDL modern, Verilog, el qual és encara molt utilitzat en la industria. El llenguatge VHDL, el qual és l'utilitzat en aquest projecte, no apareix fins al 1987 davant un requeriment del departament de defensa dels Estats Units.

En l'actualitat han anat sorgint diferents llenguatges HDL amb més o menys influencia dins la industria, podríem destacar SystemC, un conjunt de classes i macros provinents de C++ o SystemVerilog, basat en el ja esmentat Verilog i algunes extensions el qual va ser dissenyat per Synopsys i mantingut més tard per la IEEE. En aquesta última dècada, han aparegut llenguatges com Chisel, dissenyat també a Berkley, o SpinalHDL els quals al utilitzar Scala com a llenguatge base, permetent una major abstracció ja que aprofiten conceptes de llenguatges moderns com: orientació a objectes, programació funcional, *parameterized types* i *type inference*. Val a dir, que tot i que aquests últims llenguatges encara no s'han estès dins de la industria, estan agafant popularitat.

1.4 FPGA

A mitjans dels anys 80, després de l'evolució de diferents tecnologies, com PLA, PAL, GAL o CPLD, l'empresa Altera treia al mercat la primera FPGA (*Field Programmable Gate Array*). Aquestes, a diferencia de les tecnologies anteriors que estaven formades per portes AND i OR, utilitzen blocs lògics (*Logic Elements*, LE) per tal d'implementar les funcions necessàries. Encara que la majoria dels blocs lògics d'una FPGA estan construïts a partir d'una taula de veritat i un biestable que en permet emmagatzemar el resultat, sovint trobem diferents tipus de blocs lògics, en una mateixa FPGA, els quals tenen un ús més específic a l'hora d'implementar certes operacions. Concretament, en la FPGA *Cyclone IV* d'Altera, la qual s'ha utilitzat en aquest projecte, té tres tipus de blocs lògics. En la següent figura es mostra l'esquema d'un d'aquests elements lògics el qual està especialitzat en operacions aritmètiques.

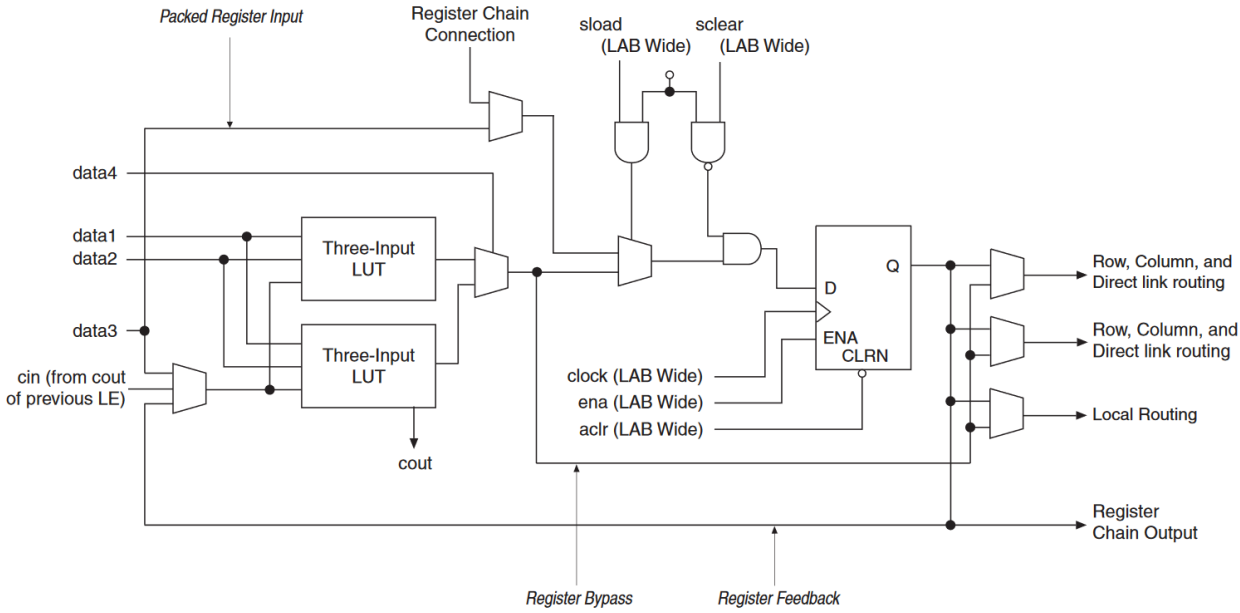


Figura 1.1: Esquema d'un dels elements lògics de la Cyclone IV

Actualment, en els xips de les FPGA també acostumen a tenir integrats funcions analògiques com DACs, generadors de senyals de rellotges, blocs de memòria o, fins i tot, com es el cas en la placa de desenvolupament de Terasic DE1-SoC, un processador de baix consum ARM.

Una de les característiques més importants de les FPGA és la seva bona relació MIPS/Watt, ja que al implementar específicament un circuit que resol el problema en qüestió, s'aconsegueixen millores elevades en rendiment i consum respecte a una arquitectura més genèrica i no optimitzada.

1.5 Estat de l'art

A l'hora d'iniciar el projecte, l'ISA RISC-V encara no s'ha acabat d'especificar, algunes de les extensions estàndards són encara susceptibles a canvis i falta documentar alguns apartats dels modes privilegiats de l'ISA. Tot i així, podem trobar per internet diferents cores RISC-V, els quals són capaços d'executar distribucions específiques de Linux. A la pàgina web de RISC-V[3] en podem trobar alguns exemples.
Una de les empresses més conegudes és SiFive els quals han dissenyat un core RISC-V anomenat Rocket[4] el qual permet executar Linux i ha sigut dissenyat utilitzant Chisel. A més a més, en la seva pàgina web[2] permeten la creació de SoC personalitzats.

Pel que fa a l'estat de l'art de projectes amb objectius similars a aquest, no s'ha trobat cap implementació que es pogués ajustar als requeriments del projecte. Doncs la majoria de projectes els quals s'han desenvolupat en un caire universitari com els processadors riscy[5], són massa complexes per a un estudiant de grau i utilitzen un llenguatge diferent a VHDL, un requeriment necessari ja que es el llenguatge HDL que es treballa durant el grau.

1.6 Entorn de treball

Per tal de desenvolupar el projecte s'han utilitzat alguns dels programes de disseny de RTL de la suite d'Intel/Altera[3] com ara:

- **Quartus**: És el programa principal i serveix de *launcher* per als altres programes de la suite. Ens permet realitzar totes les etapes necessàries per traduir el nostre codi HDL a un circuit especific per a la nostra FPGA.
- **SignalTap**: Ens permet agafar mostres del comportament de les senyals que han estat implementades en el xip de la FPGA. Ens serveix per *debuggar* el nostre disseny RTL.
- **Platform Designer**: Ens permet dissenyar un sistema, que podrem incloure dins el nostre disseny, el qual incorpora els IP cores[4] que ens ajudaran a controlar els diferents dispositius de la placa de desenvolupament.
- **Time Analyzer**: Ens permet definir les restriccions temporals del nostre disseny RTL.

Un altre software que també hem utilitzat per tal de simular el nostre disseny RTL, ha sigut el ModelSim. Un simulador de RTL el qual es pot enllaçar amb el Quartus per tal de obtenir una simulació la més concreta als aspectes que ens trobarem una vegada implementem el disseny en el xip de la FPGA.

Pel que fa a la part hardware de l'entorn de treball del projecte, després de realitzar un estudi de les plaques de desenvolupament que disposava el DAC, la DE1, DE1-SoC i DE2-115, es va decidir utilitzar aquesta última. La placa DE1, utilitzada actualment en

[2] https://www.sifive.com/
[3] Altera va ser comprada per Intel a finals del 2015
[4] Un IP core és un circuit amb una certa funcionalitat que el fabricant posa a disposició del dissenyador

l'assignatura de PEC, es va descartar ja que ha quedat descatalogada per la DE1-SoC i no es volia implementar, la possible ampliació de l'assignatura de PEC, en una placa que ja no es troba en el mercat. La segona opció, la placa DE1-SoC, no segueix la mateixa filosofia que la DE1 o DE2-115, ja que incorpora en el mateix xip de la FPGA un processador ARM dificultant l'accés a certs dispositius. Finalment es va decidir utilitzar la DE2-115 ja que era la placa de desenvolupament més complerta per a les necessitats del projecte.

La placa de desenvolupament DE2-115 compta amb una FPGA Cyclone IV, 18 interruptors, 4 polsadors, 18 LEDs vermells, 8 LEDs verds, 8 visors hexadecimals, connectors PS/2, VGA, USB, entrada d'àudio, 2 ethernets, memòries SDRAM, SRAM o flash, entre d'altres connectors o dispositius que es poden controlar directament des de la FPGA.

Capítol 2

Descripció de les fases del processador

A continuació, es detallaran les fases del projecte que, de forma incremental, han anat construint el processador. Primer de tot començarem amb la fase "Processador Unicicle"on es detalla una primera implementació de l'estructura del processador, tot seguit a la fase "Processador Multicicle"s'afegiran les diferents etapes del processador. En la fase Çontrolador de Memòria"es realitzarà la primera implementació del processador a la FPGA gràcies al controlador de memòria SDRAM que s'implementarà. En la fase "ALU"afegirem totes les instruccions aritmeticològiques d'enters que contempla l'ISA RISC-V, tot seguit, en la fase "Salts"donarem suport per executar les instruccions de salts de l'ISA RISC-V. En la fase "Dispositius d'entrada i sortida"afegirem els controladors pels diferents dispositius de la placa de desenvolupament. En la fase Ïnterrupcions"s'afegirà la lògica necessària per tal de gestionar les interrupcions, en la fase "Excepcions", molt similar a la fase anterior, s'afegiran les excepcions internes al processador, finalment en l'última fase del processador, "Mode usuari", afegirem un nou mode de privilegis al processador.

2.1 Fase Processador Unicicle

La primera implementació de l'arquitectura RISC-V, de la qual en podem trobar més informació en l'apèndix B RV32IM, consistirà en un senzill processador el qual serà capaç d'executar una sola instrucció (LUI, *Load Upper Immediate*), en un únic cicle, i ens permetrà definir els principals blocs que formaran el processador.

En aquesta primera fase del processador, al no tenir encara un controlador de memòria, s'implementarà un mòdul amb un espai de memòria de 64Kb que contindrà, de moment, les instruccions que vulguem executar. D'aquesta manera, l'ús d'un *stub* de memòria ens permetrà simular el processador amb l'eina *ModelSim*.

La jerarquia del processador tal i com es pot observar en la *Figura 1* té com a mòdul de més alt nivell, el mòdul *proc*, el qual engloba els mòduls *control unit* i *datapath*, separant així la part de control de la part de càlcul o execució del processador.

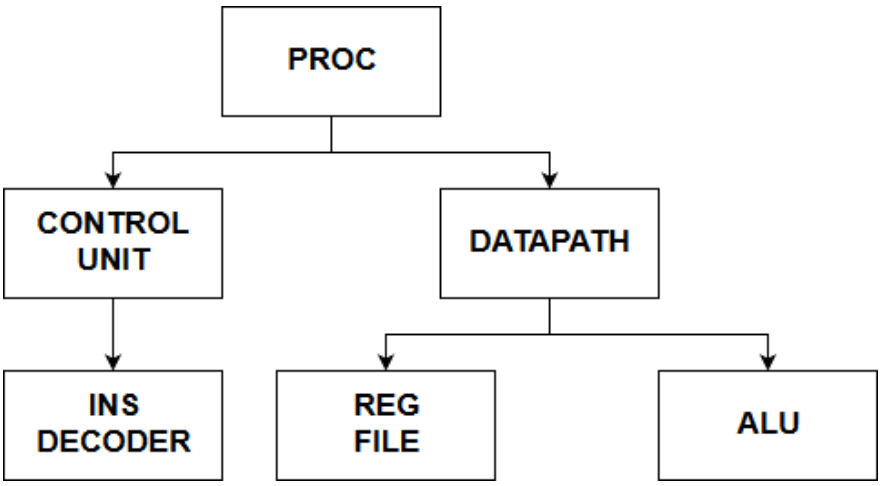


Figura 2.1: Jerarquia dels mòduls del processador en la fase Processador Unicicle

Els mòduls que s'han implementat, són els següents:

2.1.1 Proc

És el mòdul el qual interconnecta la lògica de control del processador situada en el mòdul *control unit* i la part de càlcul situada al *datapath*. Quan a la fase Controlador de Memòria tinguem ja un controlador de memòria que ens permetrà utilitzar la placa de desenvolupament, aquest mòdul serà l'encarregat d'instanciar el *core* del nostre processador al SoC.

2.1.2 Control Unit

És el mòdul més complex del processador ja que s'encarrega de governar totes les senyals d'aquest a partir de la instrucció proporcionada. També gestiona el registre del PC (*Program Counter*) el qual incrementa en 4 unitats, ja que estem en una màquina de 32 bits, a cada cicle del processador de forma indefinida.

L'ISA RISC-V no contempla cap instrucció per aturar el processador, ja que no es necessari, doncs en arquitectures modernes on un sistema operatiu s'encarrega de gestionar el hardware, aquest agafa el control quan no hi ha cap programa per executar. Al simular el processador tampoc ens serà necessari ja que podem aturar la simulació del processador just al finalitzar l'execució de l'última instrucció del codi.

2.1.3 Instrucion Decoder

El mòdul *ins decoder* s'encarrega d'obtenir la informació codificada en els 32 bits de cada instrucció i generar les senyals corresponents per tal que la part de càlcul del processador pugui executar-la correctament.

2.1.4 Datapath

És el mòdul del processador encarregat de gestionar la part d'execució de les instruccions del processador. Conté el banc de registres i l'ALU.

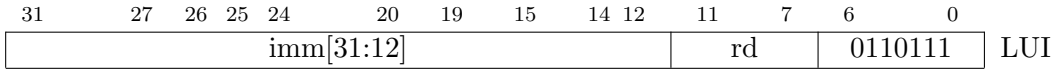
2.1.5 Regfile

És el mòdul que conté els 31 registres d'ús general del processador i un registre, el zero, el qual sempre retornarà un zero al llegir-lo i una escriptura en aquest registre serà ignorada.

2.1.6 ALU

És el mòdul que s'encarrega de realitzar les operacions aritmeticològiques que implementa el processador. En aquesta primera etapa només serà necessari que realitzi els càlculs necessaris per a la instrucció LUI.

La instrucció que s'implementarà en aquesta primera fase, tal i com s'ha comentat prèviament, és la instrucció LUI *Load Upper Immediate*, la qual carrega en els 20 bits de més pes, del bit 31 al 12, l'immediat deixant a zero els 12 bits de menor pes.



Taula 2.1: Codificació de la instrucció LUI

Per tal de comprovar la correcta implementació dels mòduls descrits anteriorment i de la instrucció LUI, s'utilitzarà els següent joc de proves:

```
1 lui t0, 0x12345
2 lui t1, 0x7FFFF
3 lui t2, 0xABC
```

Una vegada executat el codi hauríem de tenir els següents valors als registres 5, 6 i 7 del banc de registres.

Nom del registre	Número del registre	Valor
t0	5	0x12345000
t1	6	0x7FFFF000
t2	7	0x00ABC000

2.2 Fase Processador Multicicle

La majoria de processadors tenen, per tal d'augmentar el seu rendiment, un *pipeline* segmentat. Es a dir, el procés que realitza el processador des de que té la instrucció fins que acaba l'execució d'aquesta, està dividit en diferents etapes independents entre elles, gràcies a uns registres de desacoblament. Segmentar un processador comporta, a més a més, de dividir el *pipeline* en diferents etapes, implementar tota una lògica de control la qual permet gestionar correctament les dependències entre les diferents instruccions. Com que l'objectiu d'aquest TFG no és implementar un processador segmentat, no implementarem la lògica de curtcircuits i només dividirem el *pipeline* en diferents etapes aconseguint un processador multicicle.

Utilitzant de base el processador unicicle de la fase anterior, implementarem tota la lògica necessària per tal d'afegir les diferents etapes del *pipeline* i que ens permetran segmentar el processador en una futura extensió. Les etapes que implementarem, seguint l'arquitectura que es presenta en el llibre *Computer Organization and Design RISC-V edition* seran: fetch (F), descodificació de l'instrucció (ID), execució (E), memòria (M), escriptura de dades (WB). Els canvis necessaris a l'arquitectura per tal d'afegir aquestes noves etapes són, afegir un nou mòdul, el *multi*, i afegir els diferents registres de desacoblament en el *datapath* i *control unit*.

La jerarquia del processador queda de la següent forma:

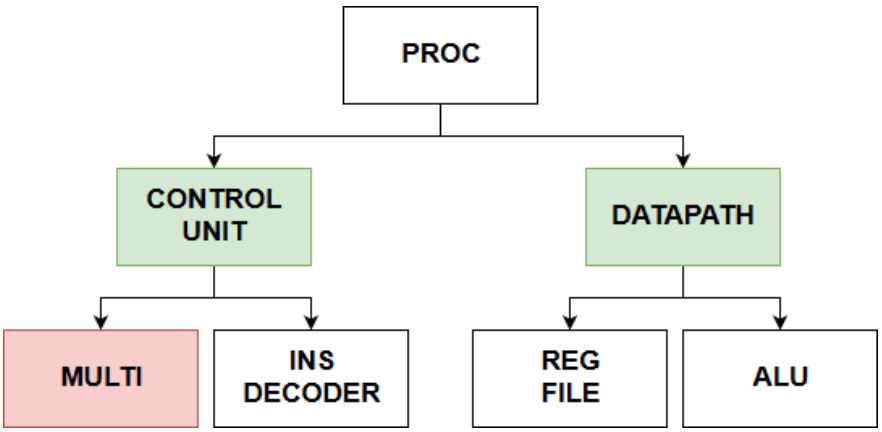


Figura 2.2: Jerarquia dels mòduls del processador en la fase Processador Multicicle

Com podem observar, els mòduls que s'han afegit o modificat són els següents:

2.2.1 Multi

És el mòdul encarregat de controlar en quina etapa es troba el processador. La seqüència que segueixen les etapes, està descrita per una màquina d'estats la qual avança al següent estat de forma incondicional a cada cicle, tret que arribi la senyal de *boot*. A la següent figura, podem observar la màquina d'estats en qüestió.

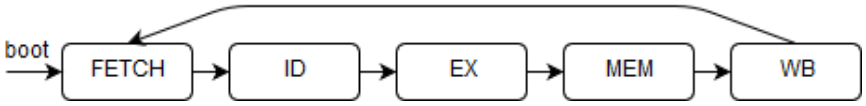


Figura 2.3: Màquina d'estats de les etapes del processador

Aquest mòdul, també s'encarregarà de filtrar aquelles senyals les quals depenen de l'estat del processador, com per exemple les senyals d'escriptura al banc de registres o d'escriptura a memòria, entre d'altres.

2.2.2 Control Unit

Al *control unit*, s'han de realitzar els següents dos canvis. El primer està relacionat amb el PC, el qual en l'etapa anterior s'actualitzava a cada cicle de rellotge, ja que l'execució d'una instrucció només trigava un cicle. Ara, al tenir un processador amb un *pipeline* de 5 etapes, el PC només s'haurà d'actualitzar al finalitzar l'etapa de FETCH.

El segon canvi que s'ha de realitzar, és afegir un registre de desacoblament, tal i com es mostren a la Figura 2.4, per tal de mantenir la instrucció llegida a l'etapa de FETCH durant l'etapa ID. Una vegada la instrucció ha estat descodificada, la unitat de control inserirà instruccions de NOP (*No OPeration*) al pipeline del processador. D'aquesta manera, ens assegurarem que el processador només executarà una única vegada la instrucció corresponent i tampoc executarà instruccions errònies.

2.2.3 Datapath

En el *datapath*, haurem d'afegir 3 registres de desacoblament, tal i com mostra la Figura 2.4, per tal de mantenir les senyals que s'han calculat a l'etapa anterior, estables en la següent etapa.

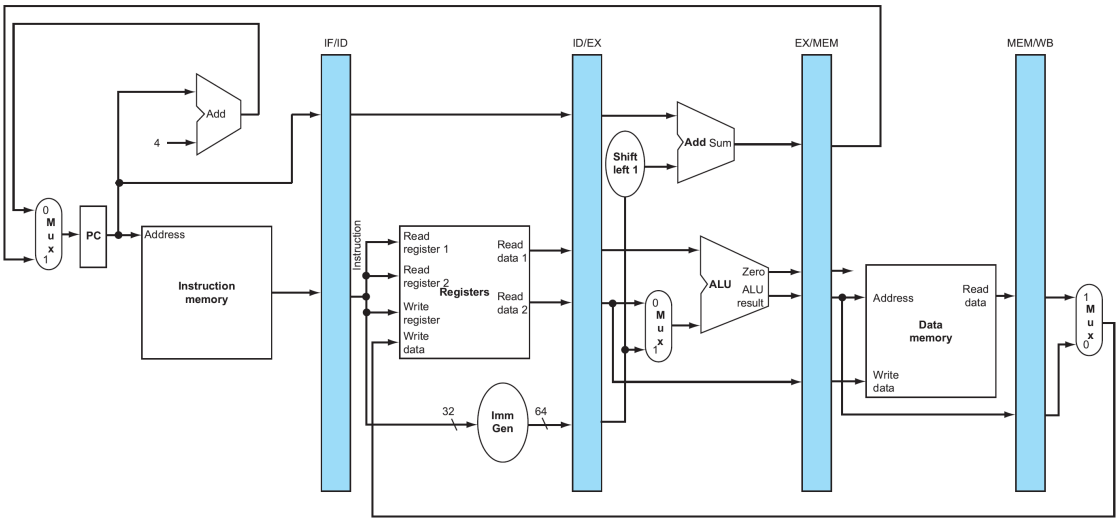


Figura 2.4: Pipeline del processador

Per tal de comprovar la correcta implementació d'aquesta etapa, podem utilitzar el joc de proves de l'etapa anterior, ja que no s'ha afegit cap nova instrucció i continuem simulant el processador amb el ModelSim.

```
1 lui t0, 0x12345
2 lui t1, 0x7FFFF
3 lui t2, 0xABC
```

Una vegada executat el codi hauríem de tenir els següents valors als registres 5, 6 i 7 del banc de registres.

Nom del registre	**Número del registre**	**Valor**
t0	5	0x12345000
t1	6	0x7FFFF000
t2	7	0x00ABC000

2.3 Fase Controlador de Memòria

Per tal d'emmagatzemar les instruccions i les dades del programa a executar, el processador necessita d'una memòria. Normalment, quan parlem de la memòria d'un processador acostumem a referir-nos a una jerarquia de memòria. Aquesta jerarquia està formada per una memòria catxe de nivell 1 per a dades i d'una altra per a instruccions, una memòria catxe de nivell 2 de dades i instruccions conjunta i, normalment, una memòria catxe de nivell 3 també conjunta de dades i d'instruccions la qual és comuna per tots els *cores* del processador. Finalment tindríem la memòria RAM i el disc.

En l'arquitectura del nostre processador només tindrem un nivell de memòria, la SDRAM, tot i que si volguéssim obtenir una millora en el rendiment del processador podríem afegir un nivell de memòria catxe, ja que les latències de la memòria SDRAM són, com veurem a continuació, bastant elevades. Afegir el controlador de memòria, ens permetrà començar a utilitzar la placa de desenvolupament per tal d'implementar físicament el nostre disseny a la FPGA.

En aquesta fase, que parteix de la base de la fase anterior, s'implementaran i modificaran els mòduls necessaris per tal de poder interactuar amb la memòria SDRAM de 128MB de la placa de desenvolupament. Concretament, s'implementaran 3 nous mòduls i s'afegiran les 8 instruccions de memòria de l'ISA RISC-V i la instrucció ADDI (*Add Immediate*) que ens ajudarà a carregar adreces de 32 bits als registres. Una vegada finalitzada aquesta fase, la jerarquia del processador serà la següent:

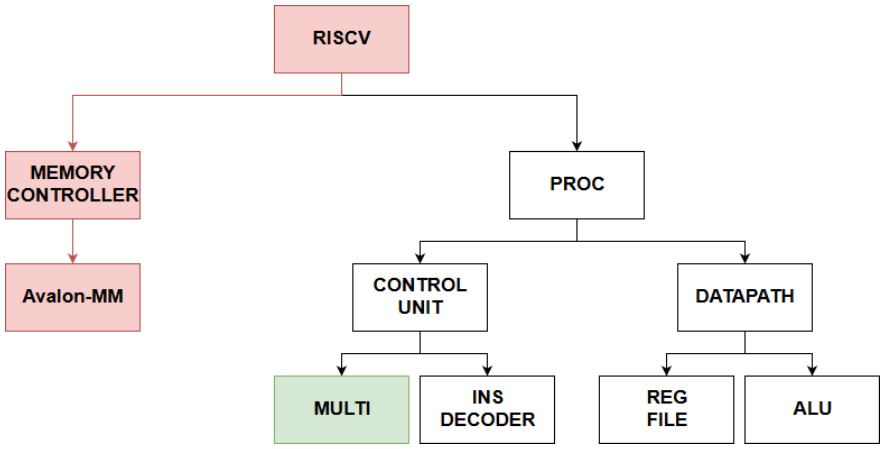


Figura 2.5: Jerarquia dels mòduls del processador en la fase Controlador de Memòria

Com podem observar, els mòduls que s'han afegit o modificat són els següents:

2.3.1 RISC-V

El mòdul *riscv*, el qual passa a ser el mòdul de més alt nivell, ens serveix d'enllaç entre el processador, els controladors dels dispositius d'entrada i sortida i els pins de la FPGA els quals estan connectats a aquests dispositius.

També s'encarrega de generar la senyal de rellotge del processador. Per tal de generar aquesta senyal, s'utilitzarà el rellotge de 50 MHz que proporciona la placa de desenvolupament dividint-lo, mitjançant un PLL (*Phase Lock Loop*), entre 4, obtenint finalment una freqüència de funcionament del processador de 12.5 MHz. S'ha escollit una freqüència relativament baixa per tal de facilitat el disseny del RTL i millorar els temps de compilació, doncs per freqüències altes les restriccions temporals són més estrictes i el procès de *fitter*, durant la compilació, és més lent de fer la seva tasca.

2.3.2 Memory Controller

El mòdul *memory controller* ens serveix d'enllaç entre les senyals del processador i el mòdul *Avalon-MM*. En aquesta primera etapa s'encarregarà de gestionar la SDRAM, però com podrem veure en les seccions IO i Interrupcions, també s'encarregarà de gestionar els dispositius d'entrada i sortida i les interrupcions d'aquests.

Aquests mòduls externs al processador, funcionaran a 50 MHZ ja que és la freqüència a la que funcionen els controladors de dispositius proporcionats per Intel a través del *Platform Dessigner*. També serà necessari afegir al disseny les restriccions temporals entre els diferents camins de les senyals per tal de que a l'hora de executar el *Fitter*, aquest pugui trobar un disseny el qual compleixi aquestes restriccions temporals. Al capítol Restriccions temporals, és dona més informació sobre aquestes restriccions temporals i com generar-les.

2.3.3 Avalon-MM

El mòdul Avalon-MM, ha sigut generat amb el *Platform Designer* i conté el IP core del controlador de SDRAM. Aquest mòdul ens servirà més endavant, per implementar tots els altres controladors de dispositius d'entrada i sortida. Al capítol dedicat a l'Avalon-MM, s'explica amb més detall el seu funcionament i la implementació interna.

2.3.4 Multi

Una vegada podem realitzar escriptures i lectures a la SDRAM, haurem de modificar el mòdul *multi* per tal d'adaptar-se als temps de resposta de la SDRAM com es pot observar en la figura 2.7 on el temps de *fetch* és de 4 cicles. Això comportarà que a partir d'ara, algunes de les etapes del processador necessitaran més d'un cicle per executar-se i haurem d'afegir una nova etapa de memòria, per permetre al processador esperar-se quan realitza una lectura d'una dada a memòria. També serà necessari emetre des del *multi*, una senyal

de bloqueig del datapath quan s'està esperant la resposta d'una lectura de memòria, ja que sinó perdríem la informació de la instrucció.

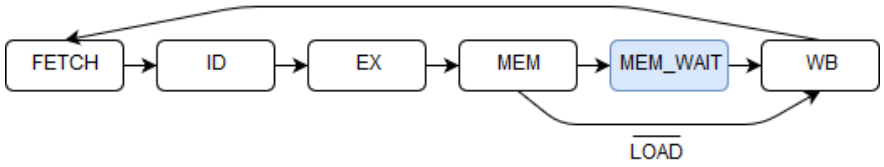


Figura 2.6: Etapes del processador

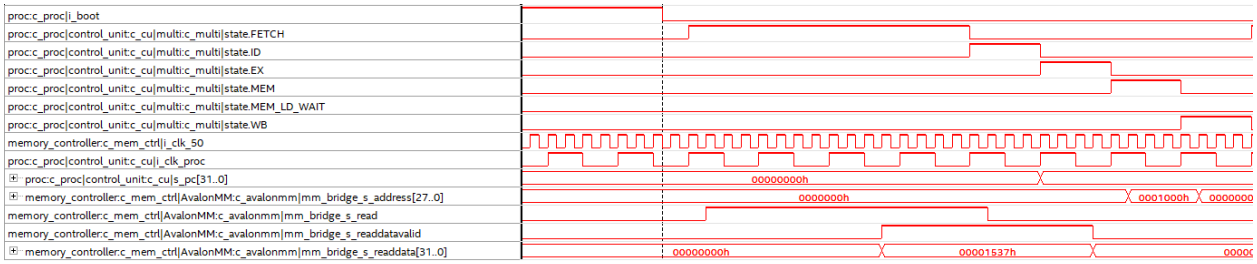


Figura 2.7: Temps de resposta de la SDRAM en cas de lectura

2.3.5 Memory Map

L'ISA RISC-V, no defineix com s'ha d'estructurar la memòria ni com s'ha de repartir les adreces entre els diferents dispositius i deixa en mans d'aquells que implementen l'arquitectura, definir el *memory map* del processador.

Com ja hem comentat anteriorment, el nostre disseny utilitzarà una memòria SDRAM de 128 MB la qual dividirem, de moment, en dues regions. Una regió de text, on anirà el codi del programa que vulguem executar i una secció de dades per emmagatzemar les dades del programa.

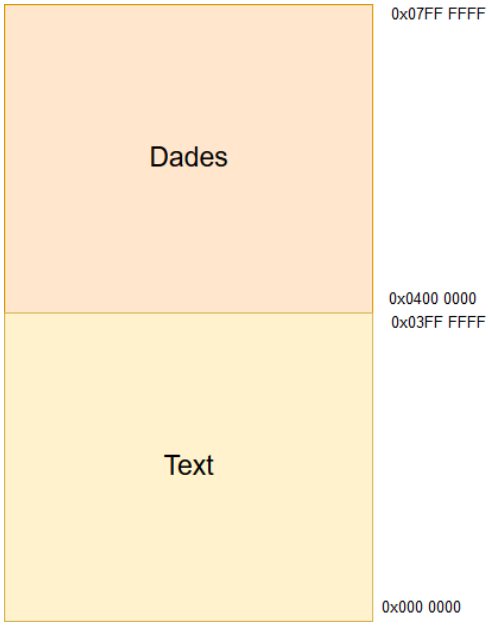


Figura 2.8: Mapeig de les adreces de la memòria SDRAM a l'espai d'adreces del processador

Les instruccions que s'implementaran en aquesta fase seran les instruccions de *load* i *store*, a nivell de *Byte*, *Halfword* i *Word*.

31 27	26 25 24 20	19 15	14 12	11 7	6 0	
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

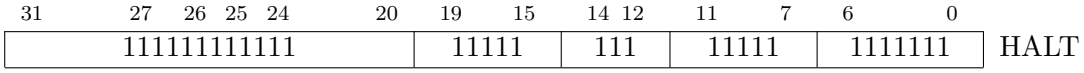
Taula 2.2: Codificació de les instruccions d'accés a memòria

I la instrucció ADDI *Add Immediate* amb la qual juntament amb la instrucció LUI, ja implementada, ens permetrà utilitzar la pseudoinstrucció LI *Load Immediate* la qual permet carregar un immediat a un registre.

31 27	26 25 24 20	19 15	14 12	11 7	6 0	
imm[11:0]		rs1	000	rd	0010011	ADDI

Taula 2.3: Codificació de la instrucció ADDI

Com que ja no estem simulant el processador, i encara no tenim implementades les instruccions de salt, necessitarem una instrucció que ens permeti aturar l'execució del processador en un moment donat. Com ja es va explicar a la secció del del processador unicicle, l'ISA RISC-V no contempla cap instrucció amb aquestes característiques, es per això que l'afegirem nosaltres de forma temporal. El nom de la instrucció serà *halt* i tindrà la següent codificació.



Taula 2.4: Codificació de la instrucció ADDI

Per tal de comprovar el correcte funcionament del processador en aquesta fase, utilitzarem el següent codi el qual realitza diferents accessos a memòria a nivell de *Byte*, *Halfword* i *Word*.

```
1  li a0, 0x04000000
2
3  li t0, 0xFFFF0000
4  li t1, 0xAA00
5  li t2, 0xAA
6
7  sw t0, 0(a0)
8  sh t1, 0(a0)
9  lw t3, 0(a0)
10 sw t3, 4(a0)
11 sb t2, 4(a0)
12
13 sw zero, 8(a0)
14 lhu t2, 4(a0)
15 lbu t1, 4(a0)
16
17 addi t2, t2, 0x100
18 addi t1, t1, 0x1
19
20 sh t2, 8(a0)
21 sb t1, 8(a0)
22 halt
```

Una vegada acabada l'execució, hauríem de tenir a les següents posicions de memòria aquests valors:

Addreça de memòria	**Valor**
0x04000000	0xFFFFAA00
0x04000004	0xFFFFAAAA
0x04000008	0x0000ABAB

2.4 Fase ALU

L'ALU (*Arithmetic Logic Unit*) és el mòdul del processador encarregat de realitzar les operacions aritmeticològiques que es contemplen a la especificació de l'ISA. És molt comú que els processadors tinguin més d'una ALU, una per cada tipus de dades a les quals es dóna suport: enters, coma flotant, doble precisió o vectors.

En aquesta fase del processador, només implementarem la ALU que realitza les operacions amb enters especificades a l'RV32I i a l'extensió M, la qual incorpora les operacions de multiplicació i divisió amb enters.

Partint de l'etapa anterior del processador, ampliarem l'ALU ja especificada afegint les noves operacions i mantenint aquelles que ja realitzava. També serà necessari afegir totes aquestes noves instruccions al *Instruction Decoder* per tal de que pugui generar els codis d'operacions de la ALU per a cada operació.

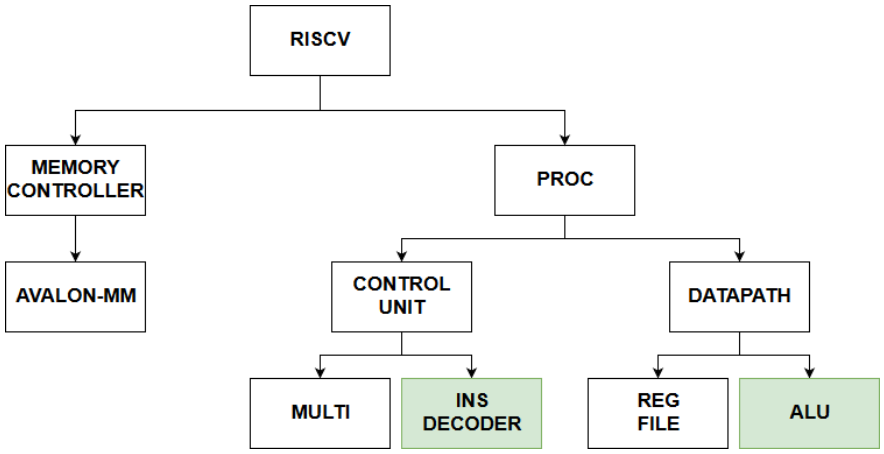


Figura 2.9: Jerarquia dels mòduls del processador en la fase Processador Multicicle

En aquesta fase, s'ha afegit quasi tota la funcionalitat del mòdul *alu*, tot i així, el seu comportament pot ser encara ampliat o modificat en les futures fases.

Les instruccions que s'han implementat en aquesta fase són les següents:

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND

Taula 2.5: Codificació de les instruccions aritmeticològiques del RV32I

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
0000001		rs2	rs1	000	rd	0110011	MUL
0000001		rs2	rs1	001	rd	0110011	MULH
0000001		rs2	rs1	010	rd	0110011	MULHSU
0000001		rs2	rs1	011	rd	0110011	MULHU
0000001		rs2	rs1	100	rd	0110011	DIV
0000001		rs2	rs1	101	rd	0110011	DIVU
0000001		rs2	rs1	110	rd	0110011	REM
0000001		rs2	rs1	111	rd	0110011	REMU

Taula 2.6: Codificació de les instruccions de multiplicació i divisió del RV32M

Sobre les operacions de divisió, caldria destacar com l'ISA RISC-V tracta les divisions per 0. On en la majoria d'arquitectures una divisió per 0 acostuma a provocar una excepció, l'ISA RISC-V, tal i com s'explica en l'apartat 6.2 del document *The RISC-V Instruction Set Manual Volume I: User-Level ISA*[1], no generarà una excepció al produir-se una divisió per 0 ja que seria l'única operació aritmeticològica que causaria excepcions. En la següent taula podem observar quin és el comportament d'una divisió per 0, o quan es produeix un *overflow*.

Condició	Dividend	Divisor	DIVU	REMU	DIV	REM
Divisió per zero	x	0	$2^{32}-1$	x	-1	x
Overflow (només amb signe)	-2^{32-1}	-1	–	–	-2^{32-1}	0

Taula 2.7: Comportament de la divisió per zero i *overflow* en divisions.

Per tal de comprovar la correcta implementació de totes les noves operacions afegides, les quals operen només amb registres, utilitzarem el següent codi:

```
li a0, 0x04000000
# ADD
li t0, 120
li t1, 80
add t2, t0, t1
sw t2, 0(a0)
# SUB
sub t2, t0, t1
sw t2, 4(a0)
# XOR
xor t2, t0, t1
sw t2, 8(a0)
# OR
or t2, t0, t1
sw t2, 12(a0)
# AND
and t2, t0, t1
sw t2, 16(a0)
# SLT
slt t2, t1, t0
sw t2, 20(a0)
# SLTU
li t3, -1
li t4, -5
sltu t2, t4, t3
sw t2, 24(a0)
# SRA
li t4, 1
li t1, 5
li t5, -40
sra t2, t4, t3
sw t2, 28(a0)
sra t2, t0, t1
sw t2, 32(a0)
sra t2, t5, t1
sw t2, 36(a0)
# SLL
sll t2, t1, t4
sw t2, 40(a0)
sll t2, t1, t3
sw t2, 44(a0)
halt
```

Una vegada executat el següent codi, hauríem de tenir en les següents posicions de memòria aquests resultats:

Addreça de memòria	**Valor**
0x04000000	0x000000C8
0x04000004	0x00000028
0x04000008	0x00000028
0x0400000C	0x00000078
0x04000010	0x00000050
0x04000014	0x00000001

Addreça de memòria	**Valor**
0x04000018	0x00000001
0x0400001C	0x00000000
0x04000020	0x00000003
0x04000024	0xFFFFFFFE
0x04000028	0x0000000A
0x0400002C	0x80000000

Per tal de comprovar les operacions amb immediats, utilitzarem el següent codi:

```
li a0, 0x04000000
# ADDI
addi t0, zero, 0xAB
sw t0, 0(a0)
# SLTI
slti t0, zero, -13
sw t0, 4(a0)
slti t0, zero, 13
sw t0, 8(a0)
# SLTIU
li t1, -50
sltiu t0, t1, 10
sw t0, 12(a0)
sltiu t0, t1, -100
sw t0, 16(a0)
# XORI
xori t0, t1, 682
sw t0, 20(a0)
# ORI
ori t0, t0, 883
sw t0, 24(a0)
# ANDI
andi t0, t0, 555
sw t0, 28(a0)
# SLLI
slli t0, t0, 5
sw t0, 32(a0)
slli t0, t0, 10
sw t0, 36(a0)
# SRLI
srli t0, t1, 8
sw t0, 40(a0)
srli t0, t1, 10
sw t0, 44(a0)
# SRAI
li t1, 0xcafe1234
srai t0, t1 10
sw t0, 48(a0)
srai t0, t0, 5
sw t0, 52(a0)
halt
```

Una vegada executat el següent codi, hauríem de tenir en les següents posicions de memòria aquests resultats:

Addreça de memòria	**Valor**
0x04000000	0x000000AB
0x04000004	0x00000000
0x04000008	0x00000001
0x0400000C	0x00000000
0x04000010	0x00000000
0x04000014	0xFFFFFD64

Addreça de memòria	**Valor**
0x04000018	0xFFFFFF77
0x0400001C	0x00000223
0x04000020	0x00004460
0x04000024	0x003FFFFF
0x04000028	0xFFF2BF84
0x0400002C	0xFFFF95FC

Finalment, el codi per provar les operacions de multiplicació i divisió, és el següent:

```
1  li a0 0x04000000
2  li t0, 14
3  li t1, 5
4  mul t2, t0, t1
5  li t1, -5
6  sw t2, 0(a0)
7  mul t2, t0, t1
8  sw t2, 4(a0)
9  mulh t2, t0, t1
10 sw t2, 8(a0)
11 mulhu t2, t0, t1
12 sw t2, 12(a0)
13 li t0, -14
14 mulhsu t2, t1, t0
15 sw t2, 16(a0)
16 li t0, 56
17 li t1, -7
18 div t2, t0, t1
19 sw t2, 20(a0)
20 divu t2, t0, t1
21 sw t2, 24(a0)
22 li t1, -9
23 rem t2, t0, t1
24 sw t2, 28(a0)
25 remu t2, t0, t1
26 sw t2, 32(a0)
27 halt
```

Una vegada executat el següent codi, hauríem de tenir en les següents posicions de memòria aquests resultats:

Addreça de memòria	**Valor**
0x04000000	0x00000046
0x04000004	0xFFFFFFBA
0x04000008	0xFFFFFFFF
0x0400000C	0x0000000D
0x04000010	0xFFFFFFFB

Addreça de memòria	**Valor**
0x04000014	0xFFFFFFF8
0x04000018	0x00000008
0x0400001C	0x00000002
0x04000020	0x00000038

2.5 Fase Salts

Per tal de trencar el seqüenciament implícit del *Program Counter*, els processador utilitzen operacions de salt les quals modifiquen el PC de forma condicional o incondicional. Aquestes operacions, permeten als programadors utilitzar rutines, bucles o branques condicionals. L'arquitectura RISC-V contempla dos tipus de salts, els salts condicionals, codificats com a instruccions de *branch*, els quals a partir d'una condició permeten modificar el PC a una posició relativa a aquest de ± 4KiB. I els salts no condicionals, codificats com a instruccions de *jump*, els quals permeten carregar qualsevol valor al PC.

En aquesta fase de la implementació, partint de la base descrita en l'apartat anterior, s'implementaran totes les operacions de salt que contempla el RV32I i la instrucció AUIPC (*Add Upper Immediate to PC*) que ens permetrà construir adreces relatives al PC. S'afegirà un nou mòdul al *datapath*, el *take branch* i s'haurà de modificar la lògica del PC en el *control unit*. L'ALU, s'encarregarà de calcular el nou valor del PC.

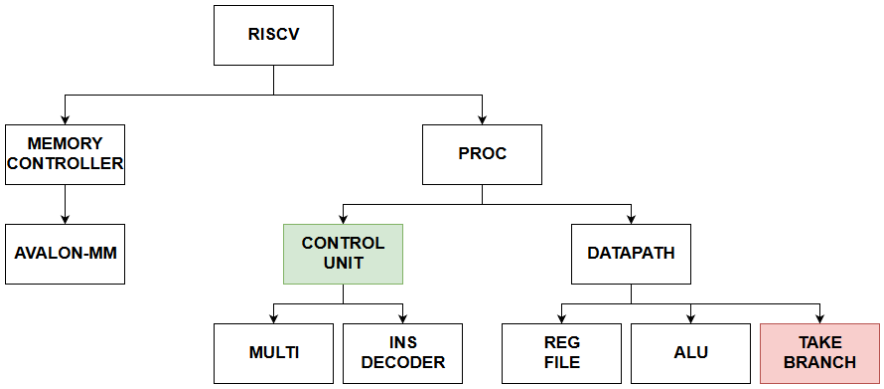


Figura 2.10: Jerarquia dels mòduls del processador en la fase Salts

2.5.1 Take Branch

És el mòdul encarregat d'avaluar les condicions dels salts condicionals i indicar al *control unit* si ha de trencar el seqüenciament implícit del PC.

En el cas que s'acabes segmentant el processador, aquest mòdul podria encarregar-se també de predir si el salt en qüestió ha de saltar o no.

2.5.2 Control Unit

En aquesta fase, hem de modificar com s'ha de tractar el PC en condició a la senyal generada pel *take branch*. En indicar-nos que s'ha de produir un salt, en l'etapa de *Write Back*, el PC passarà a tenir el valor calculat durant l'execució de la instrucció de salt per la ALU.

Les instruccions que s'implementaran en aquesta fase són les següents:

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]					rd	0010111	AUIPC
imm[20|10:1|11|19:12]					rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	BGEU

Taula 2.8: Codificació de les instruccions de salt

Una vegada implementades les operacions de *branch*, la operació de *halt* afegida per nosaltres en la fase del Controlador de Memòria la qual ens permetia aturar l'increment del PC ja no es estrictament necessaria, ja que com es pot observar en els codis de prova, un bucle infinit al final de cada programa tindrà el mateix efecte. Tot i així la mantindrem ja que ens ajudarà a *debuggar* la implementació.

Per tal de comprovar el correcte funcionament de les instruccions de *branch* i de la instrucció AUIPC en el nostre processador, utilitzarem el següent codi:

```
1  li a0, 0x04000000
2  auipc t0, 0
3  sw t0, 0(a0)
4  auipc t0, 0x1000
5  sw t0, 4(a0)
6  # BEQ
7  beqz zero, L1
8  li t0, 1
9  b L2
10 L1:
11   li t0, 2
12 L2:
13   sw t0, 8(a0)
14   # BGE
15   li t1, 5
16   bgez t1 L3
17   li t0, 1
18   b L4
19 L3:
20   li t0, 2
21 L4:
22   sw t0, 12(a0)
23   # BGEU
24   li t1, -5
25   li t2, -6
26   bgeu t1, t2, L5
27   li t0, 1
28   b L6
29 L5:
30   li t0, 2
31 L6:
32   sw t0, 16(a0)
33 # BLT
34   li t1, -5
35   bltz t1, L7
36   li t0, 1
37   b L8
38 L7:
39   li t0, 2
40 L8:
41   sw t0 20(a0)
42   # BNE
43   li t1, -5
44   li t2, -6
45   bne t1, t2, L9
46   li t0, 1
47   b L10
48 L9:
49   li t0, 2
50 L10:
51   sw t0, 24(a0)
52 END:
53   b END
```

Una vegada executat el codi, hauríem de tenir els següents resultats en les posicions de memòria corresponent:

Addreça de memòria	Valor
0x04000000	0x00400008
0x04000004	0x01400010
0x04000008	0x00000002
0x0400000C	0x000000002
0x04000010	0x000000002
0x04000014	0x000000002
0x04000018	0x000000002

Per comprovar el correcte funcionament de les instruccions de salt incondicional, utilitzarem el següent codi:

```
1 li a0, 0x04000000
2 la t1, LABEL
3 jalr t2, t1, 0
4 li t2, -1
5 LABEL:
6   sw t2, 0(a0)
7 END:
8   b END
```

Una vegada executat el codi, hauríem de tenir el següent resultat en la posició de memòria *0x04000000*:

Addreça de memòria	Valor
0x04000000	0x0000000C

2.6 Fase Dispositiu d'entrada i sortida

Per tal de rebre o transmetre informació, els processadors necessiten poder comunicar-se amb altres dispositius. Com el conjunt d'aquest dispositius és molt heterogeni, la majoria de processadors deleguen les funcions d'escriptura i lectura a aquests dispositius a un intermediari. En el cas d'Intel, els dispositius estan dividits entre el *northbridge*, per aquells dispositius més ràpids com ara la memòria RAM o els ports PCI, i el *southbridge*, on estan els dispositius més lents.

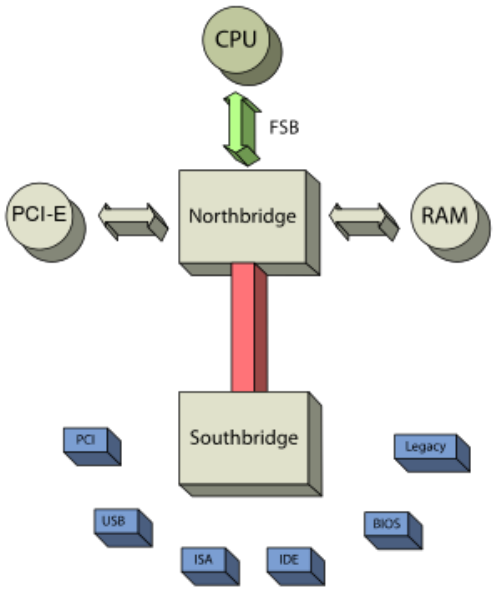


Figura 2.11: Jerarquia típica dels dispositius d'entrada i sortida en una arquitectura Intel

En aquesta fase, s'implementaran els controladors dels dispositius de la placa de desenvolupament que volem utilitzar en el nostre processador. Aquests són: els interruptors, LEDs, polsadors, visors set segments, connector PS/2 i connector VGA. Tal i com es descriu en l'especificació de l'ISA RISC-V, tots els dispositius d'entrada i sortida, estan mapejats a memòria. Això suposa que no es necessita cap instrucció especial per accedir a aquests dispositius i que el seu accés es realitzarà a través d'instruccions d'escriptura i lectura a memòria.

Per tal d'implementar els controladors de tots aquests dispositius, és tornarà a utilitzar les IPs d'Intel i s'afegiran al dispositiu de l'Avalon, tal i com és va fer en la secció del Controlador de Memòria amb la SDRAM.

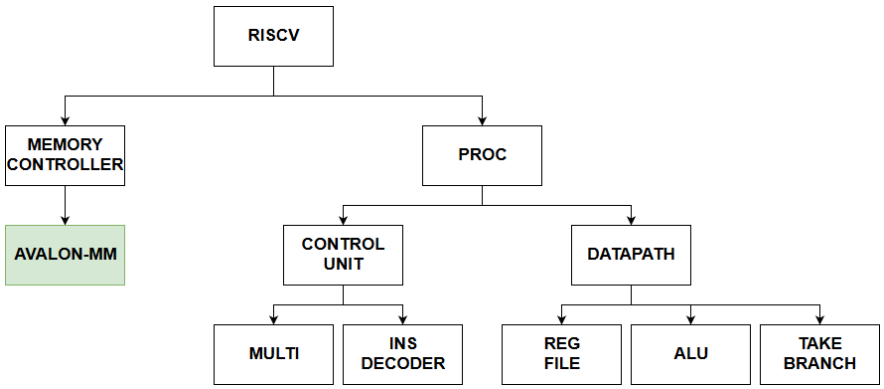


Figura 2.12: Jerarquia dels mòduls del processador en la fase Entrada i Sortida

2.6.1 Avalon-MM

En aquest dispositiu, el qual es genera automàticament a través del *Platform Designer*, s'han afegit, tal i com es descriu en l'apartat Disseny del mòdul Avalon-MM, els controladors dels dispositius d'entrada i sortida de la placa de desenvolupament.

2.6.2 Memory Map

Com s'ha comentat anteriorment, els dispositius d'entrada i sortida estan mapejats a memòria i per tant s'han d'afegir aquestes noves adreces a l'espai d'adreces del processador. S'han decidit els següents mapeigs.

Figura 2.13: Mapeig de les adreces dels dispositius IO a l'espai d'adreces del processador

Cal d'estacar que l'espai d'adreces de la VGA és tan gran ja que estem mapejant el *pixel buffer*, se'n donen més detalls a l'apartat Disseny del mòdul Avalon-MM.

Per tal de comprovar que els dispositius funcionen, així com els seus controladors, s'utilitzaran els següents jocs de proves. El joc de proves per testejar els *switch* i leds, és el següent:

```
li a0, 0x08200000 # interruptors
li a1, 0x08200030 # LEDs vermells
bucle:
  lw t0, 0(a1)
  sw t0, 0(a0)
  b bucle
```

Si tot està ben implementat, hauríem de veure l'estat dels *switch* als LEDs vermells.

Per comprovar els polsadors i els set segments, utilitzarem el següent joc de proves:

```
li a0, 0x08200010 # polsadors
li a1, 0x08200040 # 7-seg
li a2, 0x08200050 # 7-seg
la a3, FI
la a4, bucle
bucle:
  lw t0, 0(a0)
  bnez t0, B0
  li t2, 0xFFFFFFF
  sw t2, 0(a1)
  sw t2, 0(a2)
  j a3 # Salt a FI
B0:
  li t1, 1
  bgt t0, t1, B1
  li t2, 0xFFFFF80
  sw t2, 0(a1)
  sw t2, 0(a2)
  j a3 # Salt a FI
B1:
  li t1, 2
  bgt t0, t1, B2
  li t2, 0xFFFC000
  sw t2, 0(a1)
  sw t2, 0(a2)
  j a3 # Salt a FI
B2:
  li t1, 3
  bge t0, t1, B3
  li t2, 0xFE00000
  sw t2, 0(a1)
  sw t2, 0(a2)
  j a3 # Salt a FI
B3:
  li t1, 4
  bge t0, t1, FI
  sw zero, 0(a1)
  sw zero, 0(a2)
FI:
  j a4 # Repetir bucle
```

Una correcta implementació hauria d'encendre els set segments a mesura que anem premem els polsadors.

Per comprovar el dispositiu de PS/2, utilitzarem el següent codi:

```
li a0, 0x08200060 # PS/2
li a1, 0x04000000
la a2, bucle

sw zero, 0(a1)

bucle:
  lb t3, 0(a0)
  sb t3, 0(a1)
  j a2
```

Una correcta implementació hauria de mostrar a la posició de memòria *0x04000000* el valor *scan code*[1] corresponent a la tecla premuda en el teclat.

Per comprovar la correcta implementació del controlador de VGA, utilitzarem el següent codi:

```
li t0, 0x3FF # Blue
li t1, 0xFFC00 # Green
li t2, 0x3FF00000 # Red
li t3, 0 # X 0->640
li t4, 0 # Y 0->480
li t5, 640
li t6, 480
li a0, 0x08200000 # PIXEL
   BUFFER
li s5, 0x05000000

LOOP_Y:
  bge t4, t6, LOOP_END_Y
  slli a1, t4, 12 # BASE
     ADDRESS FOR THE LINE
  or a1, a1, a0
  li s1, 160
  bge t4, s1, GREEN
  mv s0, t0
  beq zero, zero, LOOP_X
GREEN:
  li s1, 320
  bge t4, s1, RED
  mv s0, t1
  beq zero, zero, LOOP_X
RED:
  mv s0, t2
LOOP_X:
  bge t3, t5, LOOP_END_X
  slli a2, t3, 2
  or a2, a1, a2 # PIXEL
     ADDRESS
  sw a2, 0(s5)
  addi s5, s5, 4
  sw s0, 0(a2)
  addi t3, t3, 1
  beq zero, zero, LOOP_X
LOOP_END_X:
  addi t4, t4, 1
  xor t3, t3, t3
  beq zero, zero, LOOP_Y
LOOP_END_Y:
  beq zero, zero, LOOP_END_Y
```

[1]Els *scan codes*, són els codis que envia el teclat PS/2. En el següent enllaç podem trobar el valor de cada tecla https://techdocs.altium.com/display/FPGA/PS2+Keyboard+Scan+Codes

Una correcta implementació, hauria de mostrar per pantalla el següent patró (Blau, Verd, Vermell). Encara que sigui un patró molt senzill, ha sigut generat pixel a pixel i per tant ens serveix de base per poder representar qualsevol tipus de gràfic.

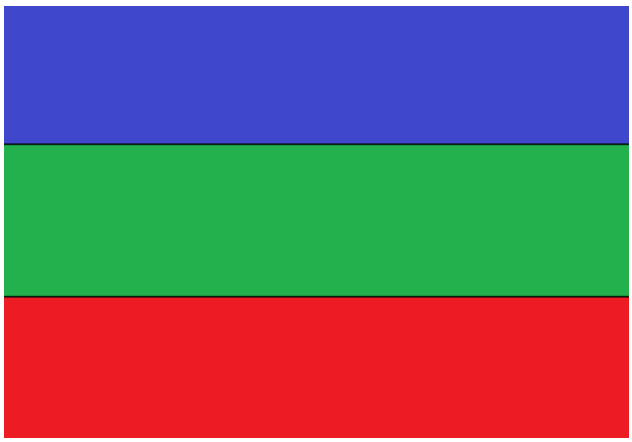


Figura 2.14: Patró de colors del codi de proves de la VGA

2.7 Fase Interrupcions

Les interrupcions són esdeveniments exteriors als processador que indiquen a aquest que algun dels dispositius que té associat necessita ser atès. L'ISA RISC-V, en el document *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*[2], descriu el gestor d'interrupcions, anomenat PLIC *Platform-Level Interrupt Controller*, que hauria d'incorporar un processador RISC-V. El PLIC, s'encarrega de rebre totes les interrupcions globals del sistema, generalment de dispositius d'entrada i sortida, i notifica a un dels cores del processador per tal de que tracti la interrupció. El PLIC, tal i com podem observar en el capítol 10 de la documentació del SoC FU540-C000 de SiFive[19], és un dispositiu mapejat a memòria.

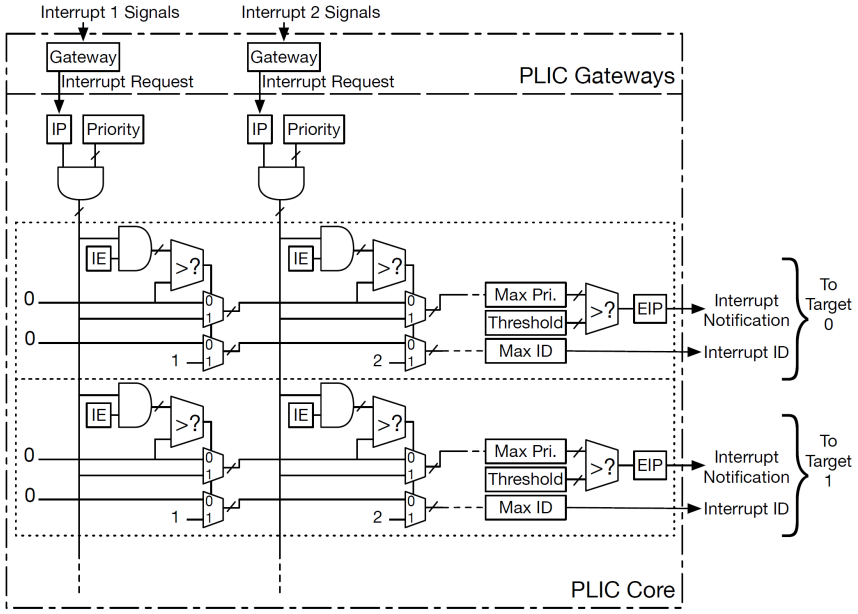


Figura 2.15: Esquema del PLIC proposat per l'ISA RISC-V

Al processador que estem dissenyant al tenir un únic core i el nombre de dispositius associats que generen interrupcions és baix, només els interruptors, polsadors, teclat PS/2 i el rellotge que incorporarem en aquesta fase, generen senyals d'interrupció. No implementarem el PLIC com a gestor d'interrupcions, ja que afegeix una complexitat que no es proporcional al processador que estem dissenyant, i per tant crearem un gestor d'interrupcions propi. També s'ha simplificat una mica el tractament de les interrupcions no permetent interrupcions niuades ni interrupcions quan el processador està en mode sistema, quan afegim en la secció Mode Usuari el mode usuari. Per tant, crearem un nou mòdul que agrupi totes les interrupcions, el *int controller*, s'hauran d'afegir els registres de sistema o CSR (*Control and Status Registers*) i les instruccions per governar-los. També s'afegirà una nova etapa de sistema i la instrucció per retornar de la rutina de tractament d'interrupcions.

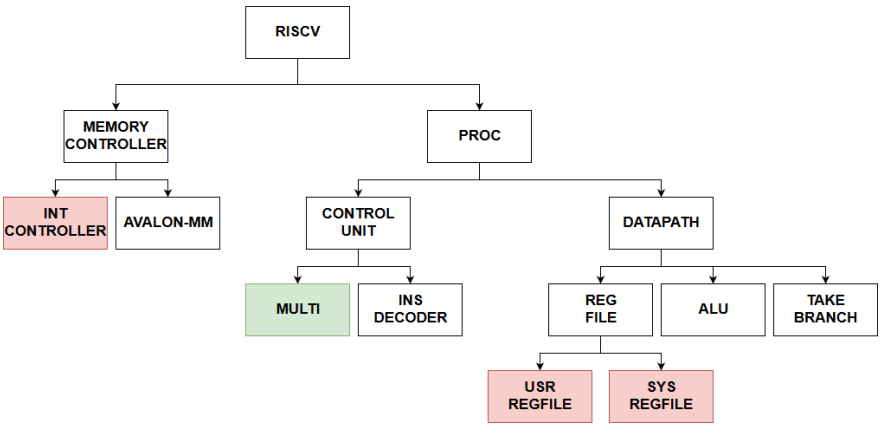


Figura 2.16: Jerarquia dels mòduls del processador en la fase Interrupcions

Com podem observar, els mòduls que s'han afegit o modificat són els següents:

2.7.1 Int Controller

És el mòdul encarregat de gestionar totes les interrupcions dels diferents dispositius associats al processador. Notifica al processador quan li arriba una interrupció i li assigna un identificador. Una vegada el processador realitza una lectura sobre el registre MCAUSE, el controlador donarà per gestionada la interrupció d'aquest dispositiu.

2.7.2 Sys Regfile

És el banc de registres del sistema, encara que en l'especificació del RISC-V es designen 4096 registres de sistema ja sigui per controlar els diferents nivells de privilegis, comptadors hardware o reservats per futures excepcions, en la nostra implementació, només existiran aquells que són realment necessaris, conservant el número de registre assignat en l'especificació. Els registres de sistema que s'implementaran són els següents:

- **MSTATUS**: És el registre encarregat de mantenir l'estat del core: nivell de privilegis, indica si les interrupcions i excepcions estan activades, etc.
- **MTVEC**: Conté l'adreça de memòria on està la rutina per tractar les interrupcions. A l'iniciar el processador contindrà l'adreça per defecte *0x00FE 0000*.
- **MTVAL**: Conte informació sobre la interrupció o excepció per ajudar al software a tractar-la.
- **MPEC**: Guarda el PC on es retornarà una vegada acabada la rutina d'interrupcions.
- **MCAUSE**: Conté l'identificador de la interrupció o l'excepció.

2.7.3 Usr regfile

És l'antic mòdul *regfile*, el qual ha estat renombrat per tal de seguir el mateix patró que el nou banc de registres de sistema i poder diferenciar-lo ràpidament i sense confusions a la implementació del processador.

2.7.4 Multi

En aquesta fase, per tal de tractar les interrupcions, s'afegirà l'etapa de sistema al processador. A aquesta etapa s'accedirà una vegada la instrucció actual ha sigut totalment executada i s'ha comunicat al processador que té una interrupció pendent.

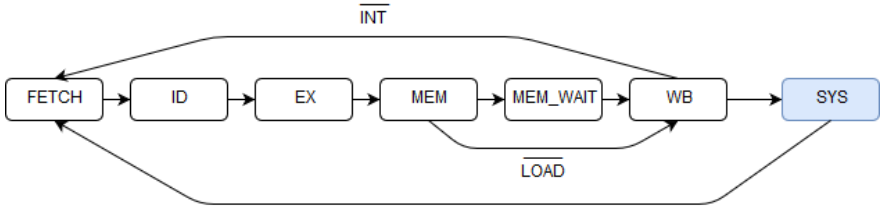


Figura 2.17: Etapes del processador en la fase Interrupcions

Durant l'etapa de sistema, el processador guardarà l'estat actual del processador, deshabilitarà les interrupcions i excepcions, guardarà el valor del pròxim PC al registre MPEC i posarà al PC el valor del registre MTVEC efectuant així un salt a la rutina de tractament d'interrupcions.

Per tal de poder llegir i escriure als registres CSR, s'implementaran les següents instruccions

31 27	26 25 24 20	19 15	14 12	11 7	6 0	
csr		rs1	001	rd	1110011	CSRRW
csr		uimm	101	rd	1110011	CSRRWI

Taula 2.9: Codificació de les instruccions de CSR implementades

S'ha decidit no implementar les següents instruccions les quals posen a 1 o 0, depenent de la instrucció, el bit indicat per *rs1* o *uimm* ja que no ofereixen una funcionalitat necesaria en el nostre processador.

31 27	26 25 24 20	19 15	14 12	11 7	6 0	
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		uimm	110	rd	1110011	CSRRSI
csr		uimm	111	rd	1110011	CSRRCI

Taula 2.10: Codificació de les instruccions de CSR no implementades

També s'ha implementat la instrucció MRET, la qual serveix per retornar de la rutina d'interrupcions.

31 27	26 25 24 20	19 15	14 12	11 7	6 0	
0011000	00010	00000	000	00000	1110011	MRET

Taula 2.11: Codificació de la instrucció MRET

Per tal de comprovar el correcte funcionament de les interrupcions en el nostre processador, utilitzarem els següents jocs de proves:

```
1  # Rutina d'interrupcions
2  csrr t0, 0x342
3  li a0, 0x0820001C
4  sw zero, 0(a0)
5  li a0, 0x08200020
6  li a1, 0x08200030
7  addi s11, s11, 1
8  sw s11, 0(a0)
9  sw s11, 0(a1)
10 mret
11
12 # Codi del programa
13 xor s11, s11, s11
14 bucle:
15   b bucle
```

Aquest codi, mostra pels LEDs el valor d'un registre el qual s'incrementa cada vegada que el processador rep una interrupció d'algun dels dispositius.

2.8 Fase Excepcions

Les excepcions són esdeveniments, molt similars a les interrupcions, però que succeeixen dins del processador degudes a l'execució d'una instrucció. L'ISA RISC-V estableix les següents excepcions en les ampliacions implementades en el nostre disseny:

- **Instrucció Il·legal**: Una instrucció és il·legal, quan aquesta no es pot executar ja sigui perquè no s'està executant amb el nivell de privilegis adequat, s'està intentant accedir a un CSR no implementat, etc.
- **Adreça d'instrucció mal alineada**: És produeix quan en un salt, la nova adreça del PC no està alineada a 4 bytes.
- **Adreça de dades mal alineada**: És produeix quan al accedir a la regió de dades de memòria l'adreça no està alineada. Es distingeix entre *loads* i *stores*.
- **Accés a regions de memòria protegides**: Es produeix quan s'intenta accedir a una regió de memòria on no es tenen el permisos necessaris.
- **Crida a sistema**: Es produeix quan s'executa la instrucció ECALL (*Environment CALL*), i serveix per fer una crida a sistema.

La gran similitud amb les interrupcions farà que en el nostre disseny el tractament de les excepcions sigui gairebé igual al de les interrupcions. Tenint aquestes, les excepcions, una prioritat superior sobre les interrupcions. S'implementarà un nou mòdul, el *exc controller* que operarà igual que el *int controller* i la instrucció ECALL.

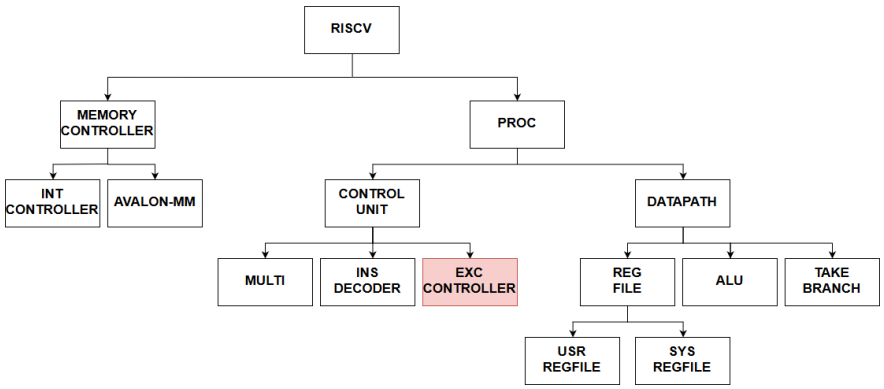


Figura 2.18: Jerarquia dels mòduls del processador en la fase excepcions

Com podem observar, els mòduls que s'han afegir o modificat són els següents:

2.8.1 Exc Controller

És el mòdul encarregat de gestionar totes les excepcions del processador i indicar-li quan l'execució d'una instrucció ha produït una excepció. Igual que en el controlador d'interrupcions, una lectura sobre el registre MCAUSE indicarà que l'excepció que s'havia emès ha estat tractada.

Com s'ha comentat anteriorment en aquesta fase, s'implementarà la última instrucció del processador la qual serveix per realitzar crides a sistema.

31 27	26 25 24 20	19 15	14 12	11 7	6 0	
000000000000		00000	000	00000	1110011	ECALL

Taula 2.12: Codificació de la instrucció ECALL

Per tal de comprovar la correcta implementació de les excepcions s'utilitzarà el següent joc de proves:

```
1  # Rutina d'interrupcions
2  csrr t0, 0x342
3  li a0, 0x08200020
4  li a1, 0x08200030
5  addi s11, s11, 1
6  sw s11, 0(a0)
7  sw s11, 0(a1)
8  mret
9
10 # Codi del programa
11 xor s11, s11, s11
12 # @ miss aligned
13 li t0, 0x00010001
14 sw t0, 0(t0)
15 lw t1, 0(t0)
16 # illegal instructions
17 csrr t1, 0xF00
18 END:
19   b END
```

El codi que s'executa, provoca tres excepcions, dos d'adreces mal alineades i una d'instrucció il·legal. Un funcionament correcte hauria de mostrar en la finalització del programa el nombre 3 en binari en els LEDs.

2.9 Fase Mode usuari

Per tal d'oferir suport als sistemes operatius, la possibilitat de virtualitzar el hardware o donar protecció als usuaris, entre d'altres, els processadors acostumen a incorporar diferents nivells d'execució els quals permeten l'execució de certes instruccions privilegiades o l'accés a regions de memòria protegides.

L'ISA RISC-V determina tres nivells d'execució, el mode Màquina on no es contempla cap restricció i es té el control absolut sobre el processador, el mode Supervisor el qual serveix per virtualitzar hardware i el mode Usuari on s'executen la majoria de programes.

En el nostre processador, s'implementaran els modes d'execució en mode màquina, el qual és el mode amb el que hem estat executant tots els nostres codis fins ara, i el mode Usuari que s'implementarà en aquesta fase i afegirà les següents restriccions:

- Serà il·legal executar instruccions que modifiquin l'estat dels CSR en mode usuari.
- Serà il·legal accedir a regions de memòria reservades per al sistema.

No s'afegirà cap mòdul nou i es mantindrà la jerarquia de la fase anterior.

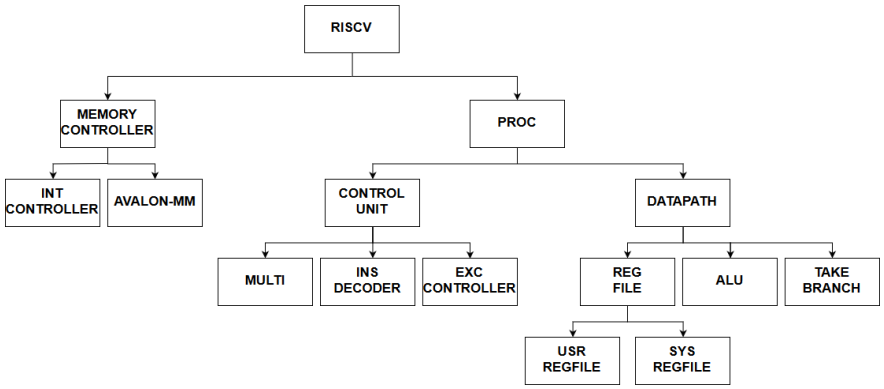


Figura 2.19: Jerarquia dels mòduls del processador en la fase Mode Usuari

2.9.1 Memory Map

Al afegir el mode usuari al processador, haurem d'afegir al mapeig de la memòria les regions de text i dades d'usuari. La SDRAM quedarà dividida de la següent manera.

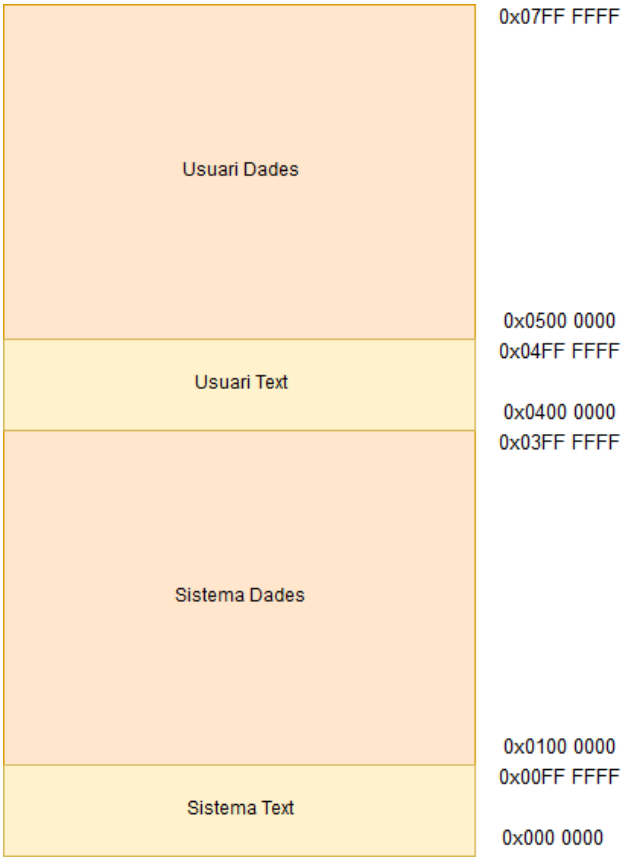


Figura 2.20: Organització de la memòria en la fase Mode Usuari

A partir d'ara si volem executar el nostre codi en mode usuari, com el processador al iniciar-se comença en mode màquina, haurem de configurar el processador i realitzar un salt a l'adreça d'inici de la regió de text d'usuari, la *0x0400 0000*.

Per tal de comprovar el correcte funcionament de la implementació d'aquesta última fase, utilitzarem el següent codi.

```
1  # Codi de sistema
2  li t0, 0x808
3  csrrw t0, 0x300, t0
4  li t0, 0x4000000
5  csrrw t0, 0x341, t0
6  mret
7
8  # Rutina d'interrupcions
9  csrr t0, 0x342
10 li a0, 0x0820001C
11 sw zero, 0(a0)
12 li a0, 0x08200020
13 li a1, 0x08200030
14 addi s11, s11, 1
15 sw s11, 0(a0)
16 sw s11, 0(a1)
17 mret
18
19 # Codi d'usuari
20 xor s11, s11, s11
21 bucle:
22   b bucle
```

Aquest programa, al inicialitzar el processador habilitarà les interrupcions i saltarà a executar el codi d'usuari. El qual mostra pels LEDs el valor d'un registre el qual s'incrementa cada vegada que el processador rep una interrupció.

Capítol 3

Implementació del processador

En el capítol anterior hem vist, quins són els mòduls que conformen el processador i els seu funcionament general. Obtenint finalment el disseny complert del processador. En aquest capítol, il·lustrarem els aspectes més importants de la implementació final del processador, senyals, bussos, etc. I de les IPs incorporades en el mòdul *Avalon-MM* les quals ens ajuden a controlar els dispositius de la placa de desenvolupament. A continuació, es mostraran fragments de codi més representatius de la implementació final. El codi sencer, es pot consultar en el fitxer adjunt a aquest document.

3.1 Unitat de control

La unitat de control, tal i com hem explicat en el capítol anterior, s'encarrega de generar les senyals de control del processador a partir de la instrucció rebuda i l'etapa en que es troba el processador. Una d'aquestes senyals és el PC i el seu comportament queda descrit per el següent fragment de codi en VHDL.

```
process (i_boot, i_clk_proc) begin
  if i_boot = '1' then
    s_pc <= RESET_VECTOR;
  elsif rising_edge(i_clk_proc) then
    if s_states = DECODE_STATE then
      if s_ld_pc = '1' then
        s_pc <= std_logic_vector(unsigned(s_pc) + 4);
      end if;
    elsif s_states = WB_STATE then
      if i_tkbr = '1' then
        s_pc <= i_new_pc;
      end if;
    elsif s_states = SYS_STATE then
      s_pc <= i_new_pc;
    end if;
  end if;
end process;
```

Com podem observar, la senyal del PC (*s_pc*), és una senyal síncrona al rellotge del processador, *i_clk_proc*, amb un reset, donat per la senyal, *i_boot*, de forma asíncrona. Aquesta senyal s'actualitzà de la següent manera. En cas de reset, se li assignarà el valor definit en la constant *RESET_VECTOR*, la qual es troba així com totes les altres constants en el fitxer *ARCH32.vhd*. Una vegada el processador està funcionant actualitzarà el valor de la senyal, augmentant-li 4 unitats (*ja que estem en una màquina de 32 bits*), de forma incondicional cada vegada que arribi un flanc ascendent de rellotge a l'etapa de descodificació de la instrucció. Com que aquesta etapa té una durada de només un cicle, el PC només s'actualitzarà una vegada. En el cas d'estar executant una instrucció de salt, en l'etapa de *Write Back*, si la senyal *i_tkbr* així ho indica, el PC passarà a tenir el valor de la senyal *i_new_pc* que contindrà el nou valor del PC calculat durant l'execució de la instrucció de salt. Finalment si durant l'execució de la instrucció s'ha produït una interrupció o excepció el PC, una vegada acabada l'execució de l'instrucció, passarà a tenir l'adreça de la rutina d'interrupcions, per defecte *0x00FE0000*. La qual està emmagatzemada en el banc de registres del sistema, en el registre *MTVEC*.

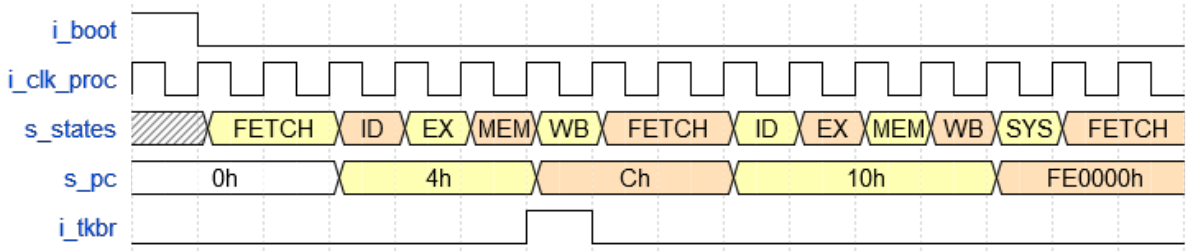


Figura 3.1: Exemple comportament de la senyal PC

La unitat de control del processador, també s'encarrega de gestionar l'estat del processador gràcies a la màquina d'estats, descrita en VHDL, que podem veure a continuació.

```
1  process (i_clk_proc, i_boot)
2  begin
3    if i_boot = '1' then
4      state <= INI;
5      s_proc_data_read <= '0';
6    elsif
       rising_edge(i_clk_proc)
       then
7      s_proc_data_read <= '0';
8      if state = INI then
9        state <= FETCH;
10     elsif state = FETCH then
11       if i_avalon_readvalid
           = '1' then
12         s_proc_data_read <=
             '1';
13       state <= ID;
14     end if;
15   elsif state = ID then
16     state <= EX;
17   elsif state = EX then
18     state <= MEM;
19   elsif state = MEM then
20     if i_ld_st = LD_SDRAM
         then
21       state <=
           MEM_LD_WAIT;
22     else
```

```
23            state <= WB;
24          end if;
25        elsif state =
              MEM_LD_WAIT then
26          if i_avalon_readvalid
                = '1' then
27            s_proc_data_read <=
                  '1';
28            state <= WB;
29          end if;
30        elsif state = WB then
31          if i_trap = '1' then
32            state <= SYS;
33          else
34            state <= FETCH;
35          end if;
36        elsif state = SYS then
37          state <= FETCH;
38        end if;
39      end if;
40    end process;
```

Màquina d'estats del processador localitzada en el mòdul multi

El processador, comença en un estat inicial, *INI*, al qual només s'hi pot arribar quan la senyal de reset està activa, d'aquesta manera aconseguim que el processador esperi en un estat on no estigui emetent lectures a la SDRAM constantment. Una vegada es posa en funcionament, cada estat anirà succeint durant un cicle amb excepció de l'estat de *FETCH* i l'estat *MEM_LD_WAIT*, al qual s'hi arriba quan es produeix una lectura a memòria. En ambdues circumstancies, per tal de canviar d'estat, haurem d'esperar a que la senyal *i_avalon_readvalid*, la qual indica que la dada llegida ja està en el bus, estigui activa. En aquest context, al canviar d'estat també li indicarem al controlador de memòria que hem llegit la dada a través de la senyal *s_proc_data_read*. En el cas de produir-se una interrupció o excepció, notificat per la senyal *i_trap*, esperarem a que s'acabi d'executar la instrucció per tal d'entrar a l'estat de *SYS*, el qual s'encarrega de preparar el processador per entrar en la rutina d'interrupcions.

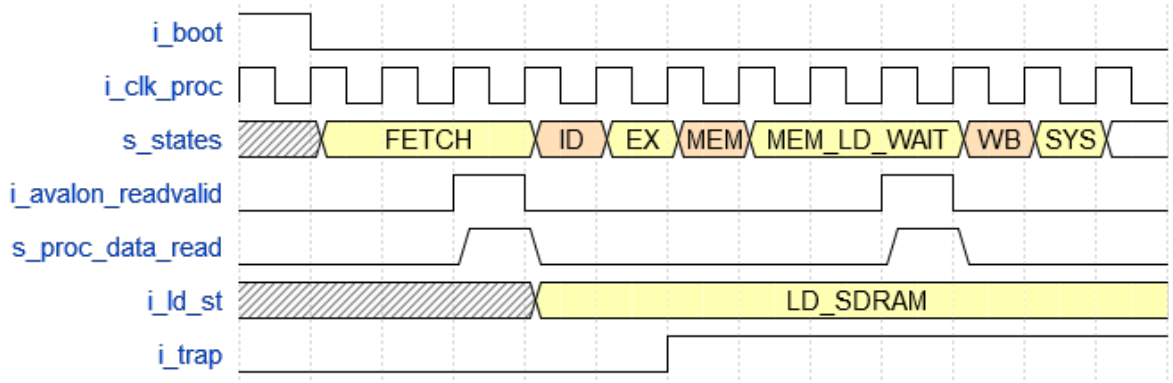


Figura 3.2: Exemple comportament de la màquina d'estats del processador

L'entitat *multi*, també s'encarrega de gestionar les senyals del controlador de memòria per tal de que es pugui efectuar el *fetch* de la instrucció, així com d'aturar l'avenç de la instrucció per el *datapath* quan s'està esperant una dada procedent de memòria o d'algun controlador.

```
o_reg_stall <= '1' when state = MEM_LD_WAIT else
         '0';

-- TO MEMORY CONTROLLER
o_ld_st_to_mc <= LD_SDRAM when state = FETCH else
              i_ld_st when state = MEM else
              IDLE_SDRAM;

o_bhw_to_mc <= W_ACCESS when state = FETCH else
            i_bhw;

o_addr_mem <= i_pc when state = FETCH else
           i_addr_mem;
```

Finalment, en la unitat de control també hi trobem el controlador d'excepcions el qual s'encarrega de gestionar-les i indicar-li al processador quan se n'ha produït una. Actualment, el processador contempla les següents excepcions: accés a memòria de dades mal alineada, accés a memòria d'instruccions mal alineada, accés a una zona de memòria protegida, instrucció il·legal i crida a sistema.

```
s_ins_addr_miss_align <= '1' when s_states = WB_STATE and i_tkbr
   = '1' and i_new_pc(1 downto 0) /= "00" else
                     '0';

s_load_addr_miss_align <= '1' when s_states = MEM_STATE and
   o_ld_st_to_mc = LD_SDRAM and o_addr_mem(1 downto 0) /= "00"
   else
                      '0';

s_store_add_miss_align <= '1' when s_states = MEM_STATE and
   o_ld_st_to_mc = ST_SDRAM and o_addr_mem(1 downto 0) /= "00"
   else
                      '0';

s_illegal_mem_access <= '1' when s_states = MEM_STATE and
   i_addr_mem < MEM_USR_CODE_INI and i_priv_lvl = U_PRIV and
   i_ld_st /= IDLE_SDRAM else
                    '1' when s_states = FETCH_STATE and
                       o_addr_mem < MEM_USR_CODE_INI and
                       i_priv_lvl = U_PRIV else
                    '0';
```

```
17 -- **INSTRUCTION DECODER**
18 o_ecall <= '1' when i_ins = PRIV_ECALL else
19           '0';
20
21 o_illegal_ins <= '1' when (o_csr_op /= CSRNOP and not (o_addr_csr
      = CSR_MSTATUS or o_addr_csr = CSR_MTVEC or o_addr_csr =
      CSR_MTVAL or o_addr_csr = CSR_MPEC or o_addr_csr =
      CSR_MCAUSE)) or (s_csr_op /= CSRNOP and i_priv_lvl = U_PRIV)
      else
22                '0';
```

Quan el controlador d'excepcions detecta un flanc ascendent en alguna d'aquestes senyals indica al processador que la instrucció que estava executant a produït una excepció. Quan el processador tracti la excepció, al efectuar una lectura al registre *MCAUSE* per tal d'esbrinar quina ha sigut la causa de la excepció, el controlador posarà en aquest registre el valor de la excepció amb més prioritat arribada fins aquell moment. En la següent taula, podem veure els identificadors de les diferents excepcions definits per l'ISA RISC-V i els nous identificadors que hem ampliat en la nostra implementació. En aquest cas, només s'ha afegit l'excepció número 24 que protegeix zones de memòria quan no es tenen el privilegis necessaris per accedir-hi. Aquesta excepció no està inclosa en el estàndard de l'ISA ja que en aquest es defineix un mòdul, anomenat PMP (*Physical Memory Protection*)[1], que s'encarrega de gestionar aquestes situacions. Per tal de que el software pugui filtrar entre excepcions i interrupcions de forma ràpida, el bit de més pes en el cas de les excepcions sempre valdrà 0, en el cas de les interrupcions, aquest bit tindrà el valor 1.

[1]Es pot trobar més informació sobre el PMP en la secció 3.6 del document *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*[2]

Interrupt	Exception Code	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved for future standard use*
0	15	Store/AMO page fault
0	16–23	*Reserved for future standard use*
0	**24**	**Access on protected memory region**
0	25–31	*Reserved for custom use*
0	32–47	*Reserved for future standard use*
0	48–63	*Reserved for custom use*
0	≥64	*Reserved for future standard use*

Taula 3.1: Valors del registre MCAUSE després d'una excepció

El controlador d'excepcions, també és l'encarregat de posar en el registre MTVAL el valor de l'adreça que ha causat l'excepció.

```
1  exceptions : process(i_clk, i_reset)
2  begin
3    if rising_edge(i_clk) then
4      if i_state_fetch = '1' and i_trap_enabled = '1' then
5        s_current_pc <= i_current_pc;
6      end if;
7      ...
8    end if;
9  end process;
10
11 o_mtval <= s_current_pc;
12 ...
```

El següent fragment de codi, mostra el comportament a l'hora de tractar l'excepció de crida de sistema. Totes les altres excepcions es tracten de la mateixa manera.

```
ecall_edge : edge_detector
port map(
  i_clk => i_clk,
  i_signal => i_ecall,
  i_data => (others => '0'),
  o_data => open,
  o_edge => s_ecall_edge
);

exceptions : process(i_clk, i_reset)
begin
  if rising_edge(i_clk) then
    if s_ecall_edge = '1' then
      s_ecall <= '1';
    end if;
    if i_exc_ack = '1' then
      elsif s_ecall = '1' then
        s_mcause <= MCAUSE_ECALL;
        s_ecall <= '0';
      end if;
    end if;
  end if;
  if i_reset = '1' then
    s_ecall <= '0';
  end if;
end process;

o_mcause <= s_mcause;

o_exc_in_order <= s_ins_addr_miss_align or s_illegal_ins or
   s_load_addr_miss_align or s_store_addr_miss_align or s_ecall
   or s_illegal_mem_access;

o_exc_trap <= o_exc_in_order and i_trap_enabled;
```

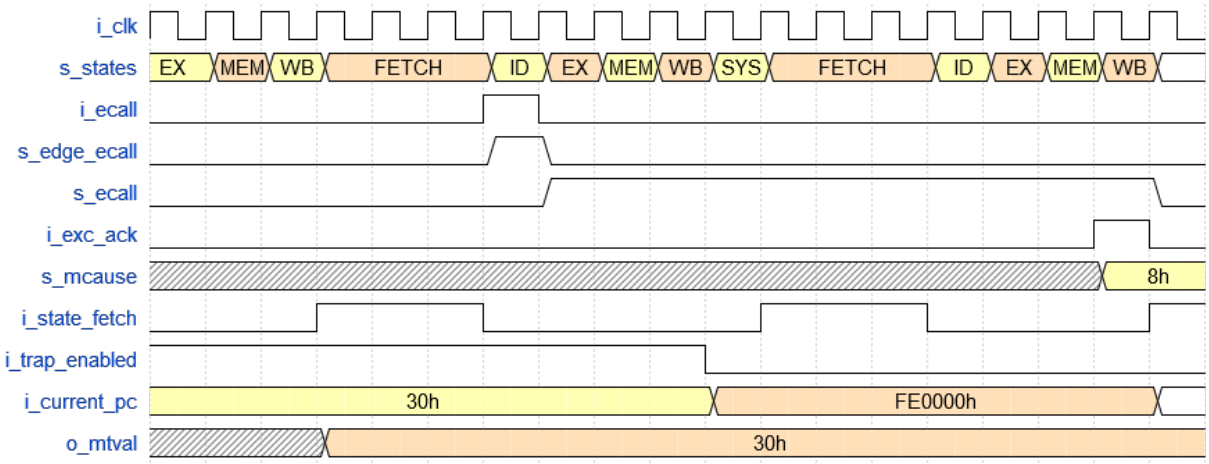


Figura 3.3: Exemple de comportament de les senyals del exception controller

3.2 Datapath

La unitat de càlcul del processador, englobada en el mòdul *datapath*, és on trobem els elements que ens permeten dur a terme la execució de la instrucció, com per exemple, la ALU o el banc de registres. En el datapath, també hi trobem la majoria dels registres de desacoblament, els quals estan definits com un bus agrupant totes les senyals.

```
signal r_id_ex    : std_logic_vector(R_DATAPATH_BUS);
signal r_ex_mem   : std_logic_vector(R_DATAPATH_BUS);
signal r_mem_wb   : std_logic_vector(R_DATAPATH_BUS);
signal r_wb_sys   : std_logic_vector(R_DATAPATH_BUS);
```

Encara que tots el busos tenen la mateixa mida, aquelles senyals que no s'utilitzin a partir d'una etapa, com poden ser les adreces dels registres font d'una operació, quedaran optimitzades pel compilador i no ocuparan espai en el xip de forma innecessària.

Com hem comentat anteriorment, la ALU s'encarrega de executar les instruccions aritmeticològiques de l'arquitectura així com també de realitzar els càlculs de les adreces quan es realitzen instruccions d'accés a memòria o de salts.

```
component alu
port (
   i_adata  : in std_logic_vector(R_XLEN);
   i_bdata  : in std_logic_vector(R_XLEN);
   i_opcode : in std_logic_vector(R_OP_CODE);
   o_wdata  : out std_logic_vector(R_XLEN)
);
end component;
```

L'entitat de la ALU està formada per les dades a operar, *i_adata* que contindrà sempre el valor d'un registre del sistema i *i_bdata*, la qual contindrà o be el valor d'un registre o el valor d'un immediat si escau. El codi de la operació que la unitat de control li ha comandat efectuar, codificat en la senyal *i_opcode*, i el resultat de la operació que es troba en *o_wdata*.

Per tal d'emmagatzemar aquests resultats, el processador compta amb 31 registres d'ús general per al programador, 32 si es té en compte el registre *zero*, el qual sempre té el valor zero. Aquests registres, es troben en el mòdul *usr_regfile* el qual està contingut dins el mòdul *regfile* que també incorpora el mòdul *sys_regfile* on estan els registres CSR.

```
1  architecture Structure of usr_regfile is
2    type reg_bank is array (R_NUM_REGS) of std_logic_vector
        (R_XLEN);
3    signal s_regs : reg_bank;
4  begin
5    process (i_clk_proc)
6    begin
7      if (rising_edge(i_clk_proc)) then
8        if (i_wr = '1') then
9          s_regs(to_integer(unsigned(i_addr_d))) <= i_port_d;
10       end if;
11     end if;
12   end process;
13
14   o_port_a <= s_regs(to_integer(unsigned(i_addr_a))) when
        unsigned(i_addr_a) /= 0 else
15           (others => '0');
16
17   o_port_b <= s_regs(to_integer(unsigned(i_addr_b))) when
        unsigned(i_addr_b) /= 0 else
18           (others => '0');
19 end Structure;
```

El banc de registres d'usuari, compta amb dos ports de lectura *o_port_a* i *o_port_b* els quals contindran el contingut del banc de registres *s_regs* indexat per *i_addr_a* i *i_addr_b* respectivament. També conté un port d'escriptura la qual es realitza de forma síncrona al rellotge del processador, sempre que la senyal d'escriptura, *i_wr*, estigui activa.

Encara que l'arquitectura RISC-V defineix espai per 4096 registres de sistema, en la nostra implementació només hem mantingut aquells registres que ens eren estrictament necessaris: *MSTATUS*, *MTVEC*, *MTVAL*, *MPEC*, *MCAUSE*. D'aquests registres, els registres *MSTATUS*, *MTVEC* i *MPEC* estan situats en el mòdul *sys_regfile*. Els altres dos, *MTVAL* i *MCAUSE* estan localitzats en els controladors d'excepcions i interrupcions. Tot i així, tal i com podem observar a continuació, s'accedeixen a través del *sys_regfile*.

```
architecture Structure of sys_regfile is
signal s_mstatus : std_logic_vector(R_XLEN);
signal s_mtvec : std_logic_vector(R_XLEN);
signal s_mpec : std_logic_vector(R_XLEN);
begin

 process(i_clk_proc, i_reset)
  begin
   if rising_edge(i_clk_proc) then
    if i_wr = '1' then
     case i_addr_d is
      when CSR_MSTATUS =>
       s_mstatus <= i_port_d;
      when CSR_MTVEC =>
       s_mtvec <= i_port_d;
      when CSR_MPEC =>
       s_mpec <= i_port_d;
      when others =>
       null;
     end case;
    elsif i_sys_state = '1' then
     s_mstatus(3) <= '0'; -- Deactivate interrupts
     s_mstatus(11) <= M_PRIV; -- Enter mode system
     s_mpec <= i_ret_pc;
    elsif i_mret = '1' then
     s_mstatus(3) <= '1'; -- Activate interrupts
     s_mstatus(11) <= U_PRIV; -- Exit mode system
    end if;
   end if;
   if i_reset = '1' then
    s_mstatus <= x"00000800";
    s_mtvec <= INT_VECTOR; -- Base address for the RSI
   end if;
  end process;

 s_mcause <= i_mcause;
 s_mtval <= i_mtval;

 o_port_a <= s_mstatus when i_addr_a = CSR_MSTATUS
   else
  s_mtvec when i_addr_a = CSR_MTVEC else
  s_mtval when i_addr_a = CSR_MTVAL else
  s_mpec when i_addr_a = CSR_MPEC else
  s_mcause when i_addr_a = CSR_MCAUSE else
  s_mtvec when i_sys_state = '1' else
  s_mpec when i_mret = '1' else
  (others => '0');

 o_trap_enabled <= s_mstatus(3);
 o_priv_lvl <= s_mstatus(11);

end Structure;
```

El mòdul *sys_regfile*, també s'encarrega de modificar els registres de sistema desactivant les interrupcions i excepcions i entrant a mode sistema quan una interrupció o excepció arriba al processador. I a tornar a restaurar l'estat anterior al retornar de la rutina d'interrupcions.

Finalment en el *datapath* i trobem el mòdul *take_branch* el qual s'utilitza en els salts condicionals i proporciona informació sobre si el salt s'ha d'agafar o no.

```
entity take_branch is
 port (
  i_adata : in std_logic_vector(R_XLEN);
  i_bdata : in std_logic_vector(R_XLEN);
  i_opcode : in std_logic_vector(R_OP_CODE);
  o_tkbr : out std_logic
 );
end take_branch;

architecture Structure of take_branch is
begin
 o_tkbr <= '1' when (i_adata = i_bdata and i_opcode = ALU_BEQ) or
  (signed(i_adata) >= signed(i_bdata) and i_opcode = ALU_BGE) or
  (unsigned(i_adata) >= unsigned(i_bdata) and i_opcode = ALU_BGEU)
     or
  (signed(i_adata) < signed(i_bdata) and i_opcode = ALU_BLT) or
  (unsigned(i_adata) < unsigned(i_bdata) and i_opcode = ALU_BLTU)
     or
  (i_adata /= i_bdata and i_opcode = ALU_BNE) or
  i_opcode = ALU_JAL or
  i_opcode = ALU_JALR or
  i_opcode = ALU_MRET else
 '0';

end Structure;
```

Tot i que aquest mòdul com podem veure realitza operacions lògiques sobre els valors dels ports d'entrada *i_adata* i *i_bdata* tal i com podria fer la ALU, aquesta no es pot utilitzar ja que està realitzant el càlcul de la adreça del salt.

3.3 Controlador de Memòria

El controlador de memòria, és un mòdul extern al processador i s'encarrega de gestionar tots els dispositius de la placa de desenvolupament oferint una interfície còmoda per al processador.

El fet que tan les diferents memòries del sistema com els dispositius d'entrada i sortida estiguin mapejats a memòria, en facilita l'ús i la seva implementació. Tal i com es descriurà en el capítol Avalon-MM el mòdul *avalon-mm*, generat a partir de les IPs d'Intel, és el que incorpora els diferents controladors dels dispositius els quals podem gestionar a través de la següent interfície.

```
component AvalonMM is
port (
   clk_clk             : in std_logic;
   mm_bridge_s_readdata : out std_logic_vector(31 downto 0);
   mm_bridge_s_readdatavalid : out std_logic;
   mm_bridge_s_writedata : in std_logic_vector(31 downto 0;
   mm_bridge_s_address : in std_logic_vector(27 downto 0);
   mm_bridge_s_write   : in std_logic;
   mm_bridge_s_read    : in std_logic;
   mm_bridge_s_byteenable : in std_logic_vector(3 downto 0);
   ...
);
end component;
```

En el cas de voler realitzar una lectura cap algun dels dispositius, hauríem d'activar la senyal *mm_bridge_s_read*, indicar l'adreça a la qual es vol accedir utilitzant la senyal *mm_bridge_s_address* i amb quina granularitat es vol accedir, *byte*, *halfword*, *word*, utilitzant la senyal *mm_bridge_s_byteenable*, on cada bit indica si s'ha d'accedir en el byte que fa referència. Una vegada la dada que volíem llegir està llesta en *mm_bridge_s_readdata* la senyal *mm_bridge_s_readdatavalid* ens ho indicaria.

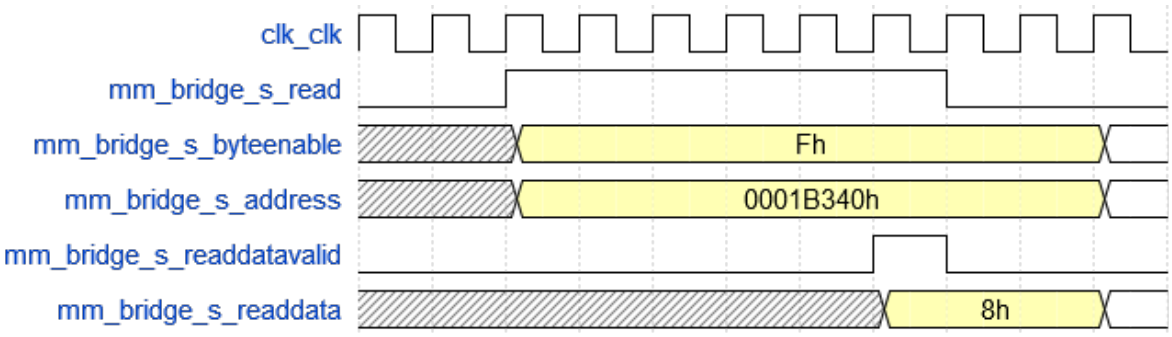


Figura 3.4: Comportament de les senyals al realitzar una lectura en el avalon-mm

Per tal de realitzar una escriptura cap alguna de les posicions de memòria, haurem d'indicar a través de la senyal *mm_bridge_s_write* que es vol emetre una ordre d'escriptura, a l'adreça de memòria indicada en la senyal *mm_bridge_s_address*, enviar la dada que es vol escriure per la senyal *mm_bridge_s_writedata* i indicar a quin nivell es vol escriure per la senyal *mm_bridge_s_byteenable*.

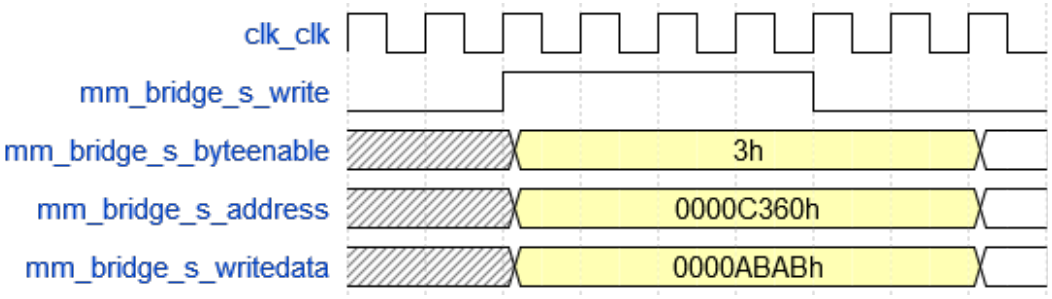


Figura 3.5: Comportament de les senyals al realitzar una escriptura en el avalon-mm

Un dels problemes que ens trobem per el fet que el controlador de memòria i el processador funcionen a freqüències diferents, 50MHz i 12.5Mhz respectivament, és que el processador no és suficientment ràpid per poder llegir les senyals del mòdul *avalon-mm*, ja que aquestes només duren un cicle del rellotge més ràpid. És per això que el controlador de memòria implementa la següent lògica per facilitar aquest mostreig al processador.

```
1  rd_hold: process(i_clk_50, s_mm_readdatavalid_edge,
       i_proc_data_read, i_reset)
2  begin
3    if rising_edge(i_clk_50) and s_mm_readdatavalid_edge = '1' and
         s_expect_readdata = '1' then
4      s_reg_readdatavalid <= '1';
5      s_reg_readdata <= s_readdata;
6    end if;
7    if i_proc_data_read = '1' then
8      s_reg_readdatavalid <= '0';
9    end if;
10   if i_reset = '1' then
11     s_reg_readdatavalid <= '0';
12     s_last_ins <= (others => '0');
13   end if;
14 end process;
15
16 o_avalon_readvalid <= s_reg_readdatavalid;
17 o_rd_data <= s_reg_readdata;
```

Al realitzar una lectura, quan el *avalon-mm* indica que la dada està llesta mitjançant la senyal *mm_bridge_s_readdatavalid*, el controlador de memòria es guarda en un registre, *s_reg_readdata*, aquesta dada i indica al processador que la dada que havia demanat ja està

llesta mitjançant la senyal *o_avalon_readvalid*, quan el processador ha llegit la dada ho fa saber al controlador de memòria gràcies a la senyal *i_proc_data_read*.

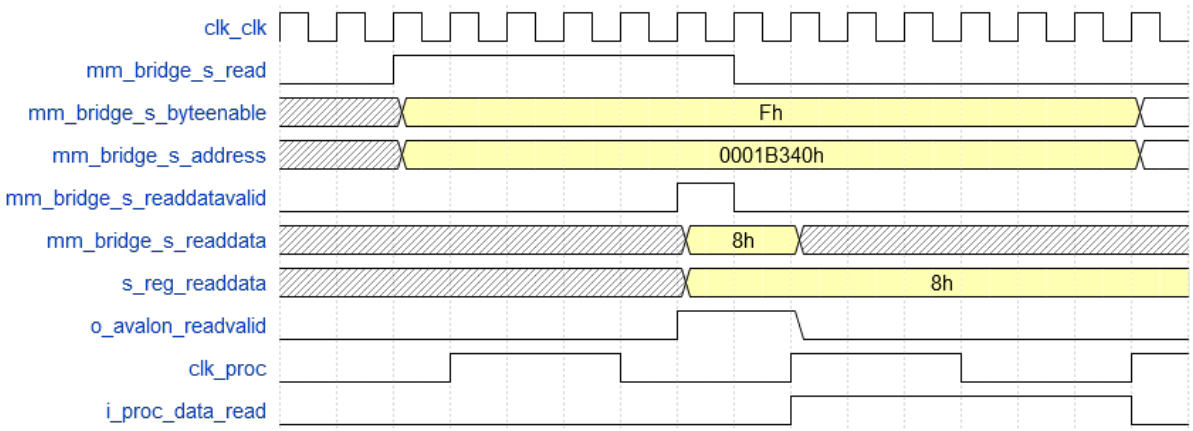


Figura 3.6: Comportament de les senyals del controlador de memòria i processador al realitzar una lectura

Per tal de controlar els diferents dispositius d'entrada i sortida, menys la VGA i PS/2 que tenen el seu propi controlador específic, s'ha utilitzat un controlador genèric anomenat PIO (*Parallel Input Output*)[20]. Aquest controlador, permet gestionar dispositius senzills com ara polsadors o LEDs mitjançant un seguit de registres mapejats a memòria. En el cas dels dispositius de sortida com ara els LEDs o els visors set segments, només ens serà necessari el primer registre del controlador anomenat *data*, el qual indica en quin estat volem que estiguin aquests dispositius. Pel que fa als dispositius d'entrada, quan aquests tenen les interrupcions activades, s'ha d'indicar al controlador a través del registre *interruptmask* quin *switch* o polsador pot produir una interrupció i netejar el registre *edgecapture*, el qual indica quin *switch* o polsador ha produït la interrupció, per tal de que aquest mateix en pugui seguir produint cada cop que el seu estat canvia. En la Figura 3.7 podem observar la configuració del PIO encarregat de controlar els polsadors.

Com que les interrupcions del processador són generades per els diferents dispositius d'entrada que aquest té associat, el controlador d'interrupcions es troba també dins el controlador de memòria. Aquest mòdul, el qual és molt similar al controlador d'excepcions, s'encarrega de gestionar totes les senyals d'interrupció dels diferents dispositius, assignar-los una prioritat i un identificador. Tal i com succeeix amb les excepcions, l'ISA RISC-V defineix uns identificadors per defecte a diferents causes d'interrupció, els quals en la nostra implementació hem ampliat amb els identificadors 16, 17, 18, corresponents a la interrupció que genera cadascun dels dispositius d'entrada/sortida. La Taula 3.2, mostra com queden distribuïts els diferents identificadors.

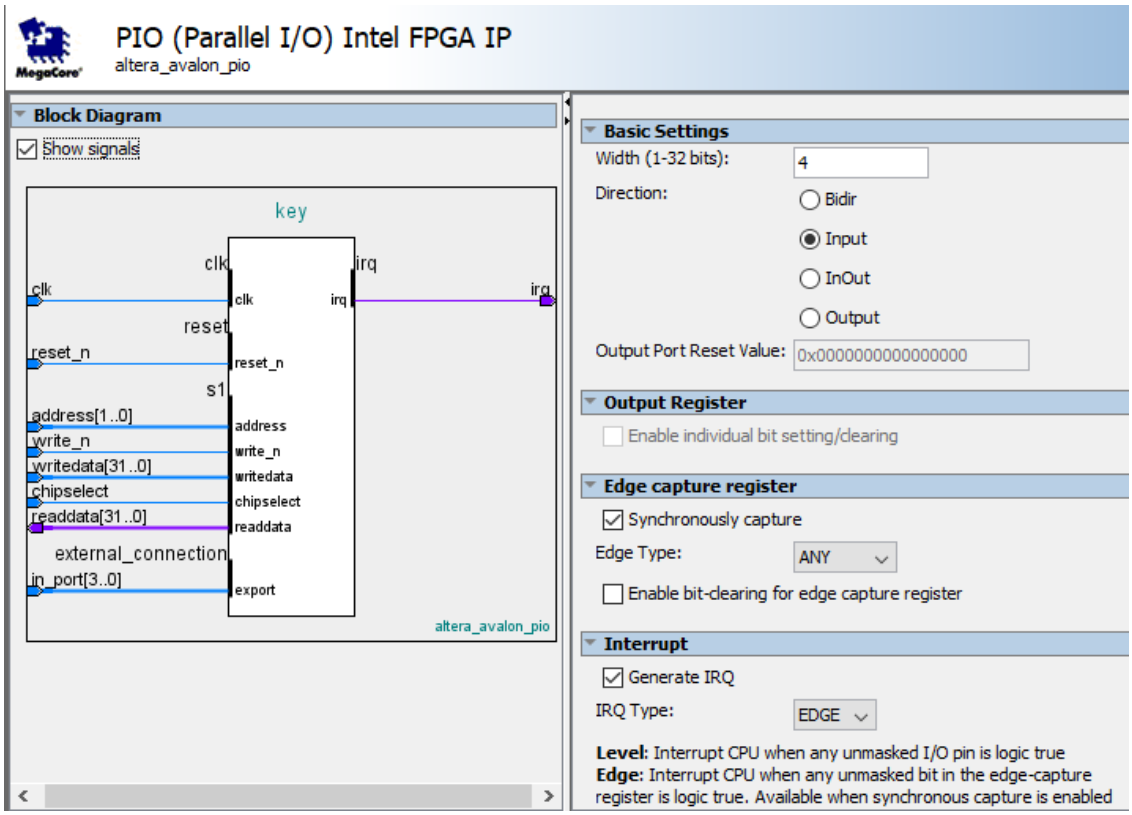


Figura 3.7: Configuració del PIO que controla els polsadors

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved for future standard use*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved for future standard use*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved for future standard use*
1	11	Machine external interrupt
1	12–15	*Reserved for future standard use*
1	**16**	**Switch**
1	**17**	**Interruptors**
1	**18**	**PS/2**
1	≥19	*Reserved for platform use*

Taula 3.2: Valor del registre MCAUSE després d'una interrupció

Capítol 4

Avalon-MM

Dins de la suite de programes per dissenyar sistemes en FPGA, Intel ofereix l'eina *Platform Dessigner* (anomenat anteriorment *Qsys*) el qual incorpora tots aquells IP Cores[20] que Intel ofereix. En el nostre disseny del processador, farem ús d'aquests IP Cores per tal de comunicar-nos amb els diferents dispositius de la placa de desenvolupament, ja que facilita el disseny i la depuració de la implementació.

4.1 IP Cores

Un IP Core és una unitat lògica reutilitzable amb un comportament definit i verificat, de la qual algú en té la propietat intel·lectual, en aquest cas Intel. La funcionalitat dels IP Cores és similar a la de les llibreries quan estem programant, estalviar temps al programador o dissenyador i ajudar-lo en aquelles tasques més complicades o tedioses.

Distingim dos tipus d'IP Cores, els *soft cores* els quals són proporcionats en un llenguatge de descripció hardware com VHDL o Verilog, tot i que a vegades també els podem trobar distribuïts en netlist. S'anomenen *soft cores* ja que permeten cert grau de modificació per ajustar-lo al disseny en el qual es vol implementar. L'altre tipus d'IP core són els *hard cores*, els quals ofereixen unes mètriques més acurades de rendiment i mida ja que no poden ser modificats. En el nostre disseny utilitzarem *soft cores*.

4.2 La interfície Avalon

La interfície Avalon[21], és un conjunt d'interfícies dissenyades per Intel les quals permeten interconnectar els diferents components disponibles en el *Platform Designer*. De les diferents interfícies que proporciona l'Avalon, en destacarem les dues que han sigut utilitzades en aquest projecte. L'*Avalon Memory Mapped Interface* i l'*Avalon Streaming Interface*.

4.2.1 Avalon Memory Mapped

És la interfície que dona nom al mòdul del processador, ja que és la dominant en el sistema. Aquesta interfície, ens proporciona un paradigma basat en escriptures i lectures amb

connexions típiques de *master* i *slave* sobre un espai d'adreces lògic. En el nostre sistema utilitzen aquesta interfície el controlador de la SDRAM, el controlador de PS/2 i els PIO (*Parallel Input Output*), encarregats del controlar els LEDs, interruptors, polsadors i visors set segments.

4.2.2 Avalon Streaming

És una interfície pensada per connexions unidireccionals, d'un dispositiu *source* a un dispositiu *sink*, amb un gran *bandwidth* i poca latència. Aquesta interfície, és la que utilitzen els diferents components que conformen el controlador de VGA.

4.2.3 Disseny del mòdul Avalon-MM

Com s'ha comentat anteriorment, el disseny del mòdul Avalon-MM s'ha efectuat en el *Platform Designer* el qual, a través de la seva interfície gràfica, permet interconnectar els diferents components i exportar aquelles senyals que han de ser governades pel processador o connectades directament als dispositius de la placa de desenvolupament.

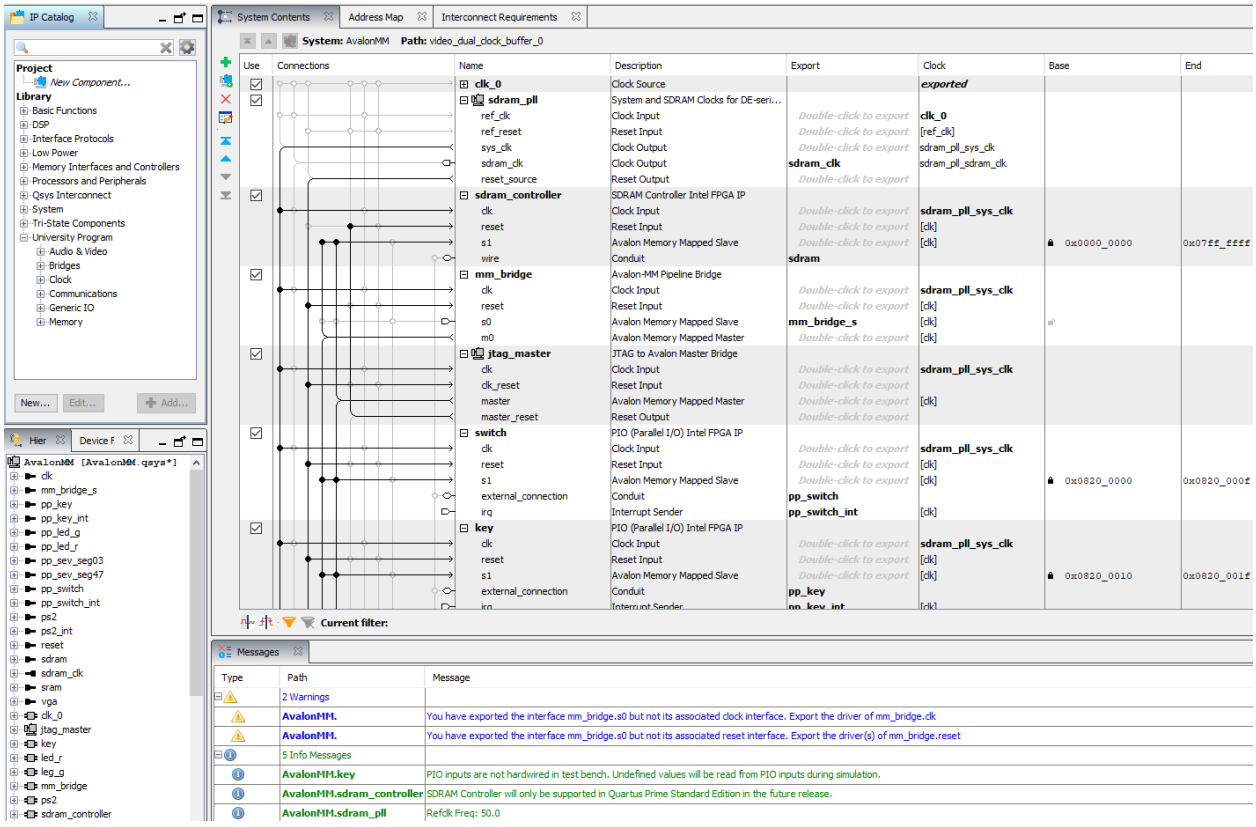


Figura 4.1: Interfície gràfica del Platform Designer

En el sistema, tal i com podem observar en la Figura 4.2 existeixen dos masters, cadascun amb el seu propòsit. El *mm_bridge* és el *master* el qual és governat pel processador i fa d'intermediari entre aquest i els diferents controladors. El *jtag_master*, serveix com a eina

de *debug* ja que ens permet, a través de la connexió JTAG i el programa *System Console*, llegir i escriure l'estat dels diferents registres dels controladors així com el contingut de les memòries, quan el processador està funcionant. Gràcies al *jtag_master*, podem carregar els programes de test a la memòria SDRAM per poder executar-los.

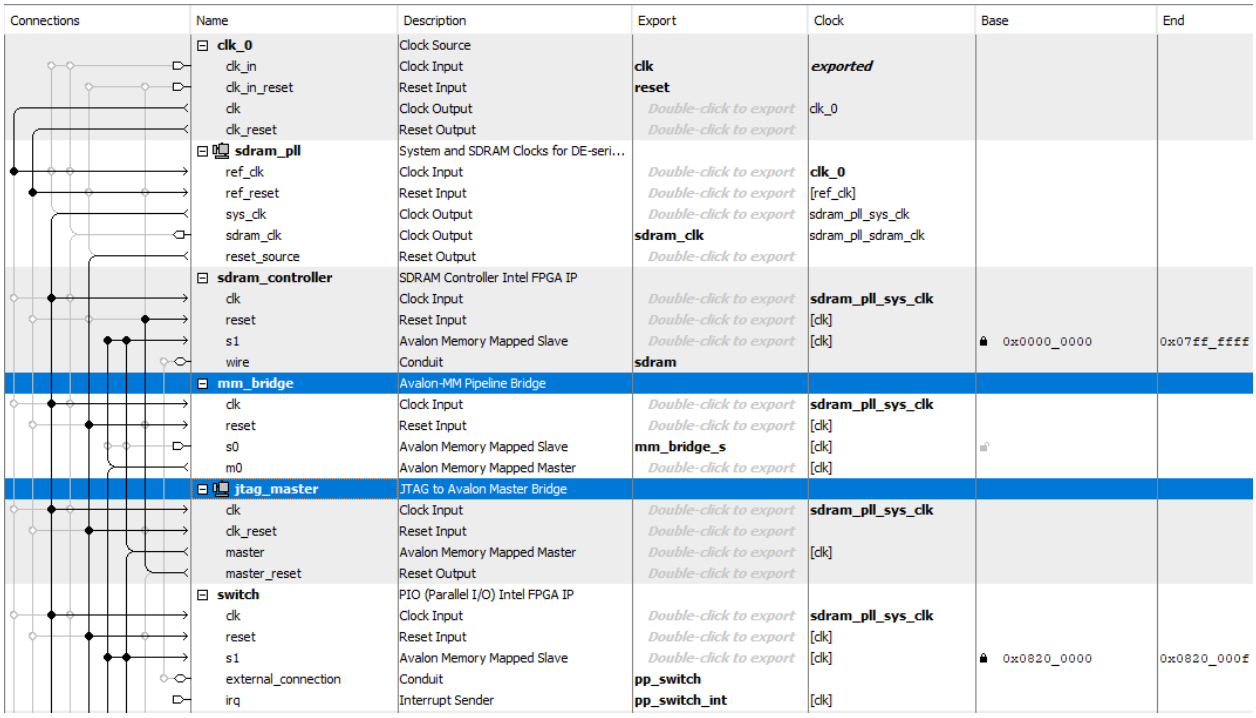


Figura 4.2: Components que realitzen de master en el mòdul Avalon-MM

Aquests *masters*, controlen els components que podem veure en la següent figura, mitjançant la interfície *Avalon Memory Mapped* que té cadascun.

Cal destacar que encara que el processador funcioni a una freqüència de 12.5 MHz, aquest mòdul té dos *clock domains*: 50MHz i 27.7MHz. El rellotge de 50 MHz, el qual és generat per el *sdram_pll* que també genera una senyal de 50MHz desfasada per a la SDRAM, és el que utilitzen tots els components del sistema menys el controlador de VGA que utilitza el rellotge de 27.7MHz generat per el *vga_pll*.

La VGA, utilitza la SRAM com a *frame buffer* ja que és una memòria molt ràpida i senzilla d'operar, no necessita d'actualitzacions constants. Aquestes característiques són necessàries per a un *frame buffer* ja que el controlador de VGA està constantment llegint d'aquest i actualitzant el contingut del monitor.

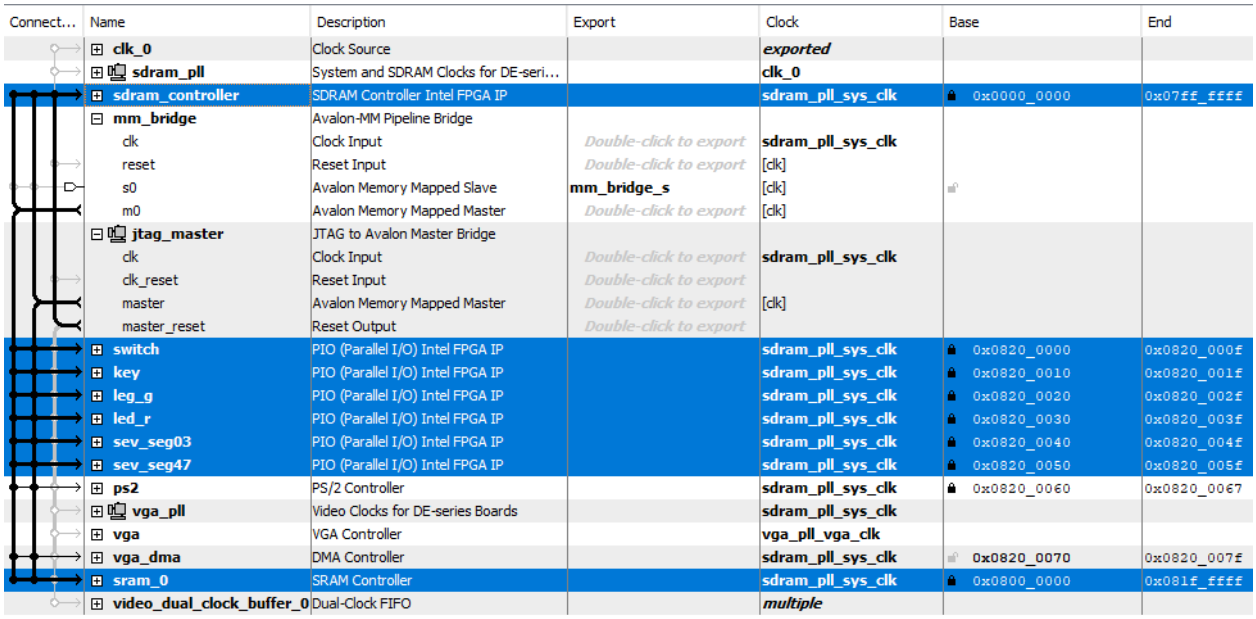


Figura 4.3: Components slaves del mòdul Avalon-MM

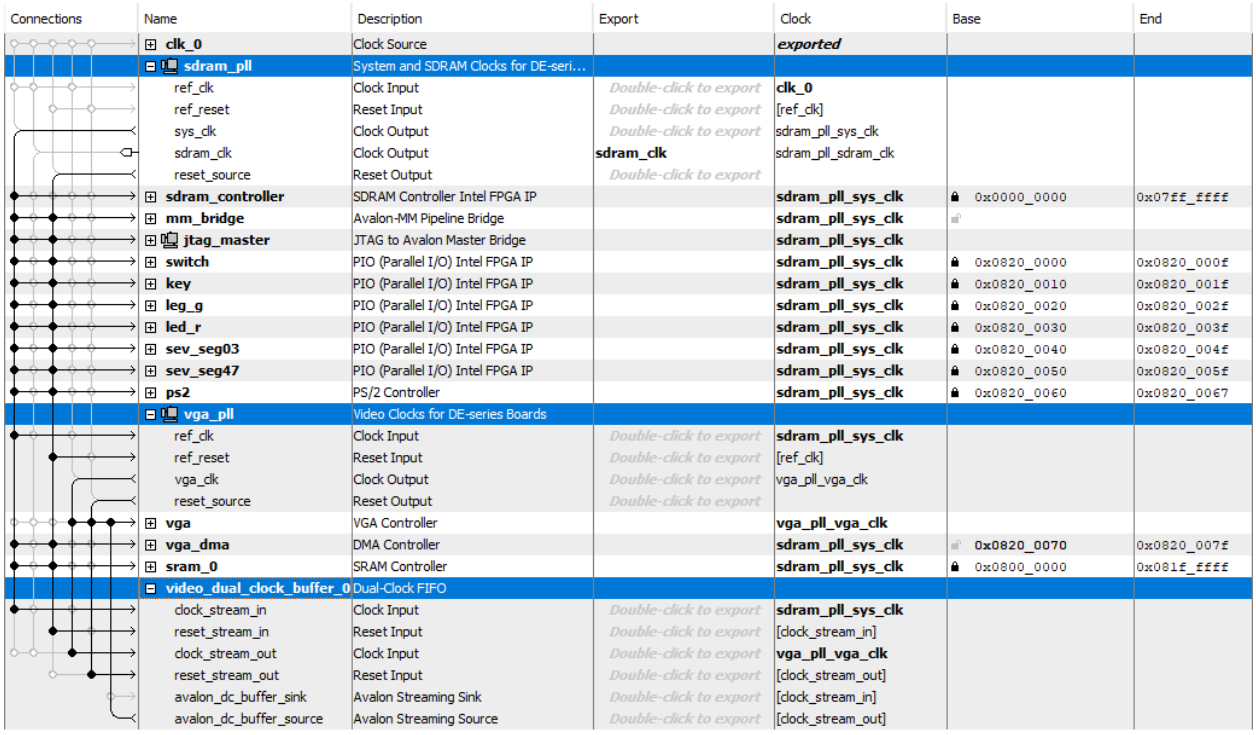


Figura 4.4: PLLs del mòdul Avalon-MM

El fet de tenir diferents rellotges en el sistema, provoca que haguem d'utilitzar *buffers* per tal d'interconnectar els dos *clock domains*. Com podem observar en la Figura 4.4 amb el *buffer* de la VGA on les dades, que son llegides de la SRAM amb un rellotge de 50 MHz, són utilitzades per el controlador de la VGA el qual funciona a 27.7 MHz.

Capítol 5

Restriccions temporals

En circuits digitals, anomenem temps de propagació al temps que triga una senyal a propagar-se d'un *flip-flop* font a un *flip-flop* de destí passant per una lògica combinacional.

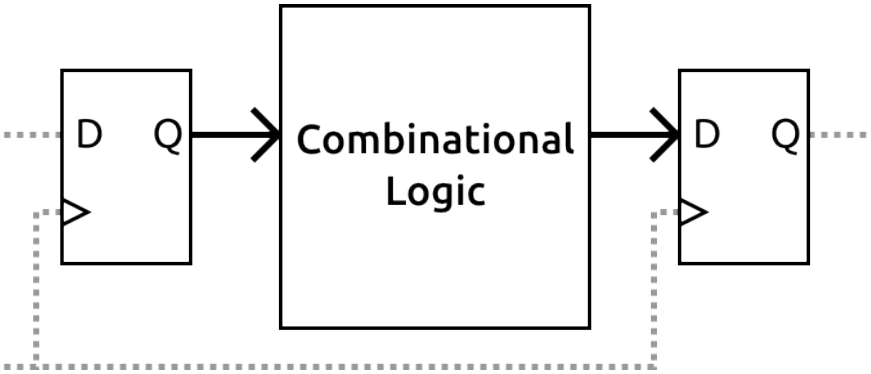


Figura 5.1: Propagació d'una senyal en un circuit digital

El fet de que aquest temps no sigui instantani, es deu al tipus de transistors (MOSFET) utilitzats per construir circuits digitals, els quals actuen com si d'un interruptor es tractés. La porta del transistor, es comporta com un condensador i triga un cert temps a carregar-se i a descarregar-se, encendre o apagar el interruptor.

Els *flip-flops* que conformen el nostre circuit digital copien el valor que tenen a l'entrada **D** cap a la sortida **Q** al arribar un flanc ascendent de rellotge, amb un cert retard al que anomenarem *D to Q*, tenint en compte que les senyals han d'estar estables durant un cert temps abans i després del flanc ascendent. Anomenem a aquest temps *setup time* i *hold time* respectivament.

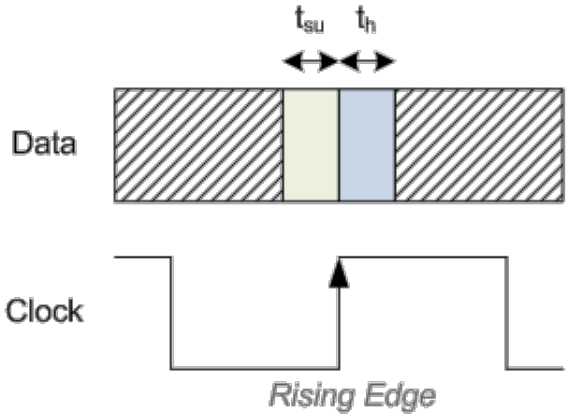


Figura 5.2: Temps de Setup i Hold

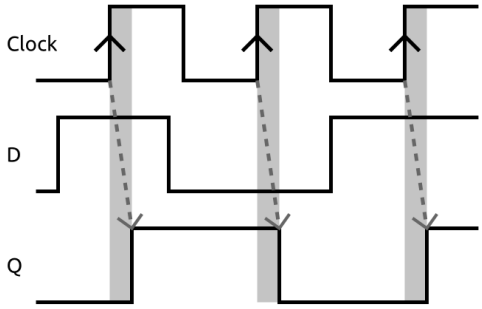


Figura 5.3: Propagació del valor de D a Q

Un altre factor que s'ha de tenir en compte, és la desviació de la senyal de rellotge degut a la seva propagació entre les diferents parts del circuit. Anomenarem a aquesta desviació *clock skew*.

Una vegada identificats tots aquests paràmetres, podem descriure la següent restricció temporal:

$$hold\ time - DtoQ \pm clock\ skew < temps\ logica\ combinacional < periode\ rellotge - DtoQ - setup\ time \pm clock\ skew$$

Com a programadors de RTL, som responsables del temps de la lògica combinacional, el qual canviarà depenent de la mida d'aquesta. Per tal de reduir aquest temps, el qual canvia a cada compilació doncs l'operació del *fitter* és un procés NP Complet, una tècnica freqüentment utilitzada és fer un *pipeline* de la lògica combinacional i realitzar-la en diverses etapes.

5.1 Timing Analyzer

Com que calcular el temps exacte és costos i canvia a cada compilació, com ja hem explicat anteriorment, el dissenyador utilitza restriccions temporals. L'eina *timing analyzer*, proporcionada en la suite de disseny de RTL d'Intel, és l'utilitzada per tal d'indicar quins són els rellotges del disseny a fi de que es puguin aplicar les restriccions temporals a l'hora de realitzar la disposició del circuit per al xip de la FPGA.

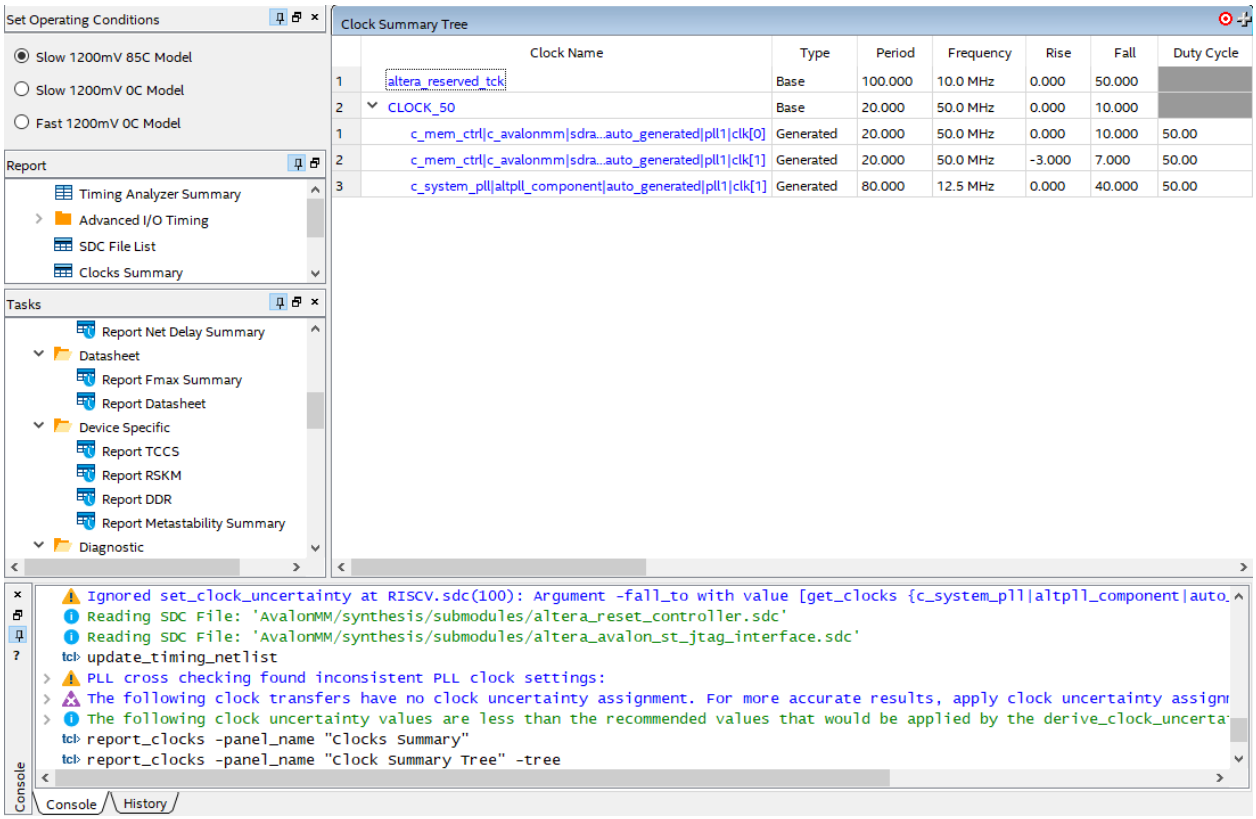


Figura 5.4: Captura del programa Timing Analyzer

Hem de tenir en compte que el programa de *fitter*, acaba quan troba la primera implementació qu compleix amb totes les restriccions temporals. Per tant com més laxes siguin les restriccions més ràpid compilarà i es sintetitzarà el codi RTL.

Si durant l'etapa del *Fitter* aquest no pogués trobar una disposició per tal de no incorre en una restricció temporal, el *Timing Analyzer* ens mostraria quina és aquesta senyal i ens donaria informació sobre el problema per tal de que el dissenyador pugui resoldre'l. En la Figura 5.5 podem observar les indicacions del *Timing Analyzer* quan una restricció temporal no es compleix.

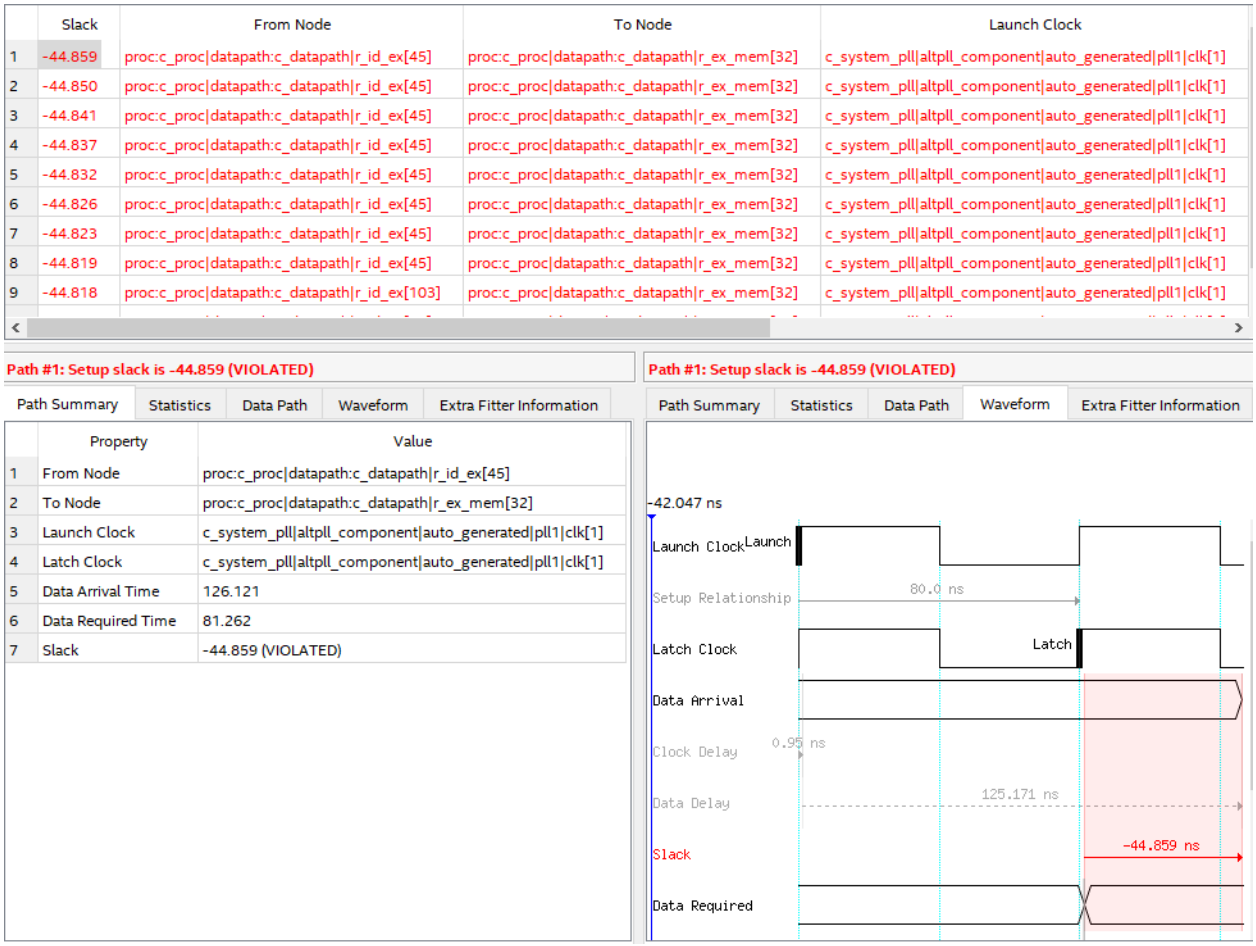


Figura 5.5: Captura del programa Timing Analyzer mostrant una restricció temporal no complerta

Capítol 6

Planificació temporal

En aquest apartat, es detallarà la planificació temporal que es va realitzar al començament del projecte per tal d'estudiar-ne la viabilitat temporal.

Es preveu detallar la planificació i justificació de les tasques a dur a terme per tal de completar el projecte amb èxit. També s'establirà quin és l'ordre de realització d'aquestes tasques i quines dependències hi ha entre elles. S'inclouran també quins són els recursos necessaris per a dur a terme les tasques, quines possibles desviacions poden haver en el projecte i quin impacte poden tenir sobre el resultat final.

6.1 Descripció de les tasques

Tot seguit es detallaran les tasques que conformen el projecte en ordre cronològic. Es detallarà breument el recursos necessaris per dur a terme cada tasca i quines dependències té, més endavant, però, és detallaran més profundament aquests aspectes així com l'estimació temporal de cada tasca.

6.1.1 Familiarització amb l'ISA RISC-V i l'arquitectura dels processadors

Abans de començar a implementar o definir l'arquitectura del nostre processador, primer cal documentar-se sobre quines són les característiques i especificacions d'aquesta nova arquitectura i quines són les idees que s'han de tenir en compte a l'hora de dissenyar un processador. Per tal de documentar-se sobre l'ISA RISC-V s'utilitzaran dos documents, la pròpia especificació de l'arquitectura[1] i el llibre *Guia práctica de RISC-V*[8]. Respecte a la documentació sobre arquitectura de computadors, s'ha consultat el llibre *Computer Organization and Design RISC-V edition*[9].

En aquesta etapa també s'investigaran les implementacions de codi públic ja fetes de l'arquitectura RISC-V.

Els recursos que s'utilitzaran per aquesta tasca són un ordinador per tal de visualitzar la documentació esmentada.

6.1.2 Gestió del projecte

Aquesta tasca contempla tota la feina d'autoaprenentatge i lliuraments del mòdul de Gestió de Projectes, 4 en total. La dedicació temporal s'estima que sigui d'unes 47 hores, aquestes hores queden repartides de la següent forma:

- **Abast del projecte i contextualització**: 15 hores.
- **Planificació temporal**: 5 hores.
- **Gestió econòmica i sostenibilitat**: 7 hores.
- **Presentació del projecte i document final**: 20 hores.

Per a la realització d'aquesta tasca es farà ús dels següents recursos: *TeXstudio*[6], el cercador *DuckDuckGo* i *Gantter*[7]

6.1.3 Anàlisi del projecte

En aquesta tasca, que té com a dependència la tasca anterior, es valorarà juntament amb el director del projecte quins són els aspectes necessaris a desenvolupar durant el projecte i quin abast tindrà la implementació de l'ISA, es a dir, quines instruccions i mòduls de tots els que conformen l'ISA s'implementaran al projecte. També s'acordaran els diferents controladors que s'hauran d'implementar.

6.1.4 Entorn de treball i placa de desenvolupament

Aquesta tasca, la qual no té cap dependència, servirà al desenvolupador per preparar l'entorn de desenvolupament, es a dir, descarregar i instal·lar tot el software necessari per al projecte: els *binutils* de RISC-V[13], el simulador RARS[10], el programa de desenvolupament per a la FPGA Quartus[11], el simulador de RTL ModelSim[12] i el programa de control de version Git[14].
En aquesta etapa, el desenvolupador també farà una tria sobre quina és la placa més adient per al projecte entre aquelles que el DAC pot proporcionar. Per tal de fer aquesta tria s'estudiarà l'arquitectura de cada placa i es realitzaran diferents tests.

Per aquesta etapa s'utilitzara un ordinador de sobretaula amb un sistema operatiu Windows 10 amb connexió a internet. S'utilitzaran les eines esmentades anteriorment per tal de fer les proves i l'eina Vim per tal d'editar els codis. També s'utilitzaran les plaques de desenvolupament de la casa Terasic que es testejaran.

6.1.5 Disseny i implementació

Aquesta etapa, que depèn de la tasca entorn de treball i placa de desenvolupament, es concentren totes les subtasques necessàries per la implementació i disseny del projecte, ja que una sola tasca seria inviable. A continuació es llisten les subtasques que divideixen aquesta tasca més gran:

- **Processador unicicle**: S'implementaran els principals mòduls que definiran l'estructura del processador. Aquest processador, el qual serà més aviat un esquelet, serà capaç d'executar una única instrucció. Cada instrucció serà executada en un sol cicle de rellotge.

- **Processador multi-cicle**: S'implementaran les etapes que acostumem a trobar en els processadors comercials. Es a dir, l'execució d'una instrucció es realitza a través de diversos cicles de rellotge del processador, dividint l'execució de la instrucció en etapes.

- **Memòria**: En una primera part s'implementaran les instruccions d'accés a memòria, tot seguit s'implementarà el controlador de memòria per una de les memòries de la placa de desenvolupament. El fet de poder utilitzar el dispositiu de memòria de la placa ens permetrà començar a testejar la implementació del processador en la FPGA de la placa de desenvolupament.

- **ALU**: En aquesta etapa s'implementaran totes les operacions aritmeticològiques de l'ISA.

- **Salts**: S'implementaran les operacions de trencament de seqüència implícit que contempla l'ISA.

- **Entrada/Sortida**: En aquesta etapa s'implementaran el controladors per al dispositiu de 7 segments i per a la pantalla LCD de la placa així com també totes les instruccions que permeten interactuar amb els dispositius d'entrada i sortida.

- **PS2 i VGA**: En aquesta etapa s'implementaran els controladors per a la gestió del dispositiu PS2 per tal de poder connectar un teclat i el controlador de VGA per tal de connectar la placa a una pantalla. També es faran els canvis necessaris a l'arquitectura.

- **Interrupcions i excepcions**: En aquesta etapa s'implementaran els mòduls d'interrupcions i excepcions del processador. Aquesta etapa sigui segurament la més difícil del projecte i per tant a la que se li hagin de dedicar més hores.

- **Mode sistema**: En aquesta etapa s'afegira la protecció de mode sistema. Obtenint dos modes de funcionament del processador, un mode usuari el qual té restringides certes operacions i regions de memòria i un mode sistema sense restriccions.

- **Operacions de coma flotant**: En aquesta etapa s'afegirà suport hardware per l'execució d'instruccions de suma i resta de coma flotant.

Per realitzar aquestes tasques s'utilitzaran els recursos esmentats en la secció d'entorn de treball i placa de desenvolupament.

6.1.6 Memòria final

La tasca final constarà de la redacció de la memòria del projecte i es prepararà el material necessari per la defensa davant del tribunal.

Per dur a terme aquesta tasca s'utilitzarà un ordinador amb el programa TeXstudio.

6.1.7 Dependències entre les tasques

Tasca	Dependències
Familiarització amb l'ISA RISC-V i l'arquitectura de processadors	-
Gestió del projecte	Familiarització amb l'ISA RISC-V i l'arquitectura de processadors
Anàlisi del projecte	Familiarització amb l'ISA RISC-V i l'arquitectura de processadors
Entorn de treball i placa de desenvolupament	Anàlisi del projecte
Processador unicicle	Entorn de treball i placa de desenvolupament
Processador multi-cicle	Processador unicicle
Memòria	Processador multi-cicle
ALU	Processador multi-cicle
Salts	ALU
Entrada/Sortida	Memòria + Salts
PS2 i VGA	Entrada/Sortida
Interrupcions i excepcions	PS2 i VGA
Mode sistema	Salts
Operacions de coma flotant	Processador multi-cicle
Memòria final	Interrupcions i excepcions + Mode sistema + Operacions de coma flotant

Taula 6.1: Taula de dependències entre les tasques del projecte

6.1.8 Previsió temporal

En la següent taula es detallen totes les tasques amb la seva previsió temporal.

Tasca	hores
Familiarització amb l'ISA RISC-V i l'arquitectura de processadors	12
Gestió del projecte	55
Anàlisi del projecte	8
Entorn de treball i placa de desenvolupament	25
Processador unicicle	10
Processador multi-cicle	8
Memòria	45
ALU	12
Salts	15
Entrada/Sortida	12
PS2 i VGA	25
Interrupcions i excepcions	60
Mode sistema	32
Operacions de coma flotant	28
Memòria final	35
Total	382

Taula 6.2: Previsió temporal de les tasques del projecte

6.2 Recursos

Tot seguit es detallen els diferents tipus de recursos necessaris per al projecte.

6.2.1 Recursos Humans

- **Director**: Seguiment i supervisió dels objectius del projecte
- **Desenvolupador**: S'encarregarà de tot el projecte.

6.2.2 Recursos hardware

- **Ordinador**: Intel Core i5-5600 i 8 GB de RAM, amb pantalla de 21.5", teclat i ratolí.
- **Placa de desenvolupament**: DE2-115 amb la FPGA Cyclone IV d'Altera.

6.2.3 Recursos Software

- **Sistema Operatiu**: S'utilitzarà un Windows 10 i una virtualització d'un sistema Ubuntu per tal d'utilitzar les binutils de risc-v.
- **Control de versions**: S'utilitzarà Git.
- **Editor**: S'utilitzarà Vim.
- **Software d'altera**: S'utilitzarà la versió 18.0 del Quartus.
- **Planificació**: S'utilitzarà l'eina Gantter.

6.3 Pla d'acció i valoració d'alternatives

Tal i com indica la normativa del TFG de la FIB, el primer dia per les defenses orals del projecte és el dia divendres 01 juliol 2019, per tant la memòria ha de ser lliurada el dia 24 de juny de 2019, es a dir, una setmana abans de l'inici de les defenses. Per tant fora bo que el projecte quedes tancat una setmana abans d'entregar la memòria.

Tot i haver repartit el temps pensant amb quines tasques són les que poden comportar més problemes a l'hora d'implementar-les, hi ha poc marge de maniobra. Es per això que en certs aspectes del projecte, aquells en els quals diverses solucions són possibles es plantegen les següents alternatives:

- **Controlador de memòria**: La placa de desenvolupament amb la que es dura a terme el projecte compta amb una memòria SDRAM i una memòria SRAM. L'objectiu principal del projecte és utilitzar la memòria SDRAM ja que compta, significativament, amb més espai. Tot i així el controlador de memòria per a una SDRAM és molt més complex que el controlador de memòria d'una SRAM, es per això que en el cas de no aconseguir implementar el controlador de memòria a través del *Platform Designer*, s'utilitzarà la memòria SRAM per dur a terme el projecte amb la qual ja estem familiaritzats.

- **Controladors VGA i PS2**: L'objectiu del projecte és implementar, utilitzant el *Platform Designer*, un controlador de VGA i PS2 per tal d'interactuar amb el processador. Si no s'aconseguís implementar satisfactòriament els controladors, s'utilitzarien les implementacions ja fetes d'aquests protocols per a l'assignatura de PEC.

- **Coma flotant**: Aquesta última etapa del projecte, és una etapa la qual no es estrictament necessària per al funcionament del processador, es per això que podríem prescindir d'ella en el cas que fos necessari per tal d'acabar el projecte en els terminis establerts.

Tots aquests canvis estarien consensuats amb el director del projecte.

Gràcies a aquesta planificació la qual té en compte possibles entrebancs durant el projecte, es garanteix la finalització d'aquest en els terminis establerts.

Capítol 7

Gestió econòmica

En aquest capítol, es detallarà l'informe econòmic realitzat al inici del projecte per tal d'avaluar-ne els costos i la seva viabilitat econòmica.

7.1 Pressupost dels recursos

En aquesta secció es mostra una taula per cada tipus de recurs utilitzat amb les seves característiques més importants, com ara el preu, vida útil, amortització i unitats. Una vegada fet l'inventari de tots els tipus de recursos, humans, software i hardware, es revisaran els imprevistos i les contingències i es calcularà el pressupost total.

Recurs Humà	Dedicació(hores)	Preu/hora (€/h)	Cost total(€)
Director del projecte	56	50[17]	2800
Desenvolupador	382	15	5730
Total			**8530**

Taula 7.1: Costos del recursos humans.

En relació a les hores de dedicació del director del projecte, les quals es fan conjuntament amb el desenvolupador, s'ha fet una estimació tenint en compte les reunions per setmana que es realitzen(1), la durada d'aquesta reunió(2 hores) i la duració del projecte (7 mesos).

Recurs Software	Preu(€)	Unitats	Vida útil	Amortització(€/h)
Windows 10	139	1	2 anys	0.032
Ubuntu 18.0	0	1	2 anys	0
RISC-V Binutils	0	1	-	0
Quartus	0	1	-	0
Vim	0	1	-	0
Git	0	1	-	0
TexStudio	0	1	-	0
Total				**0**

Taula 7.2: Costos dels recursos software.

Recursos hardware	Preu(€)	Unitats	Vida útil	Amortització(€/h)
Ordinador	1200	1	4	0.14
DE2-115[15]	309	1	10	-
Total	**1509**			**0.14**

Taula 7.3: Costos dels recursos hardware

Per tal de calcular l'amortització d'aquells recursos hardware i software s'ha utilitzat la següent fórmula: $\frac{Preu}{Dies(264)*Hores(8)*Anys}$

7.2 Costos indirectes

Encara que la major part del projecte es desenvolupi a casa del desenvolupador i per tant no hi hagi cap cost relacionat amb el lloguer d'una oficina si que computarem com a cost indirecte la despesa en electricitat del hardware utilitzat i el cost del desplaçament per a assistir a les reunions amb el director.

Cost Indirecte	Preu per més(€/mes)	Duració(mesos)	Cost Total(€/h)
Electricitat	5	7	35
Tarja T-10 1 Zona	10	7	70
Total			**105**

Taula 7.4: Estimació de costos indirectes

El consum del conjunt ordinador més monitor és d'uns 850W i el consum de la placa de desenvolupament és d'uns 50W aproximadament. El preu del kiloWatt d'electricitat és d'uns 0.12€/kWh[18].

7.3 Imprevistos i contingències

Una vegada feta la planificació temporal, podem observar com les etapes de Memòria i Interrupcions i excepcions són aquelles les quals tenen destinades més hores degut a la seva dificultat i per tant més propenses a patir imprevistos en forma de desviacions temporals, les úniques que afectarien al cost final del projecte. Ja que ambdues etapes podríem considerar que tenen un risc moderat, és raonable fixar un percentatge de contingència del 10% sobre el cost de la tasca.

Degut a la envergadura del projecte, s'estima un percentatge total de contingència d'un 8% sobre els costos directes del projecte, que es preveu adient i no s'espera sobrepassar. Les desviacions en les tasques, poden comportar un augment de, 3 hores addicionals del director i 7 hores addicionals per al programador, fet que suposaria un total de *3 hores x 50 €/h + 7 hores x 15 €/h = 255 €*. Aquesta xifra queda en la seva totalitat inclosa en el 8% de contingència dels costos del projecte i es reflecteix en la taula de pressupost del projecte.

7.4 Pressupost final

En aquesta secció es presenta la taula amb el pressupost definitiu del projecte, incloent totes les previsions per cada tipus de recurs i tasca.

	Unitats	**Amortització(€/h)**	**Temps(hores)**	**Total(€)**
Costos directes				**65.57**
Gestió del projecte			**55**	**9.5**
Ordinador	1	0.14		7.7
Windows 10	1	0.032		1.76
Anàlisi del projecte			**8**	**1.4**
Ordinador	1	0.14		1.12
Windows 10	1	0.032		0.25
Entorn i placa de desenvolupament			**25**	**4.3**
Ordinador	1	0.14		3.5
Windows 10	1	0.032		0.8
Disseny i desenvolupament			**247**	**44.37**
Ordinador	1	0.14		34.6
Windows 10	1	0.032		8
Imprevistos				1.77
Memòria	10% del cost de la tasca: 7.75			0.77
Interrupcions i excepcions	10% del cost de la tasca: 10.32			1
Memòria final			**35**	**6**
Ordinador	1	0.14		4.9
Windows 10	1	0.032		1.12
Costos indirectes				**105**
Costos de recursos humans				**8530**
Total acumulat				**8577**
Contingència	**8% del total acumulat**			**686**
Total sense *IVA*				**9263**
Total amb *IVA*	**mes 21% de 9263€**			**11208**

Taula 7.5: Pressupost total del projecte

7.5 Control de gestió

Una de les desviacions més freqüents en els projectes informàtics és la durada de les diferents etapes que conformen el projecte, ja que durant el desenvolupament poden sorgir imprevistos. Es per això que és important portar un registre específic de les hores que s'han invertit en una tasca i contrastar-ho amb les hores estimades en la planificació. Per tal de reduir les desviacions del pressupost establert, una bona pràctica és realitzar la comparació del recursos consumits amb els recursos estimats una vegada es completa cada tasca, així com també l'assoliment dels objectius. El fet de portar un seguiment estricte comportarà una detecció prematura d'un possible desviament i per tant serà més fàcil corregir-ho. Aquest seguiment s'assolirà ja que setmanalment es faran reunions de seguiment.

Davant les dificultats que puguin aparèixer, aquest projecte no necessitarà de recursos addicionals als que s'ha previst, ja que la possibilitat de l'ús de hardware addicional és inexistent i tot l'entorn software és d'ús gratuït. En el cas d'haver-hi desviacions en els recursos humans, com s'ha esmentat amb anterioritat, la contingència serà capaç d'assolir el cost monetari que impliquin aquestes desviacions. És per aquesta raó que s'ha afegit un 8% de contingència al projecte, per cobrir riscos existents. Les desviacions esmentades es calcularan amb els següents indicadors:

- **Desviacions per conceptes**:
  - Desviacions en la realització de tasques en cost: $(costEstimat - costReal) * horesReal$
  - Desviacions en la realització de tasques en hores: $(horesEstimades - horesReals) * costReal$
  - Desviacions en recursos en cost: $(costEstimat - costReal) * costReal$
- **Desviacions totals**:
  - Desviacions totals en la realització de tasques: $costEstimatTotal - costRealTotal$
  - Desviacions totals de recursos: $costEstimatTotalRecursos - costRealTotalRecursos$
  - Desviacions totals de costos fixes: $costEstimatTotalFix - costRealTotalFix$

Capítol 8

Sostenibilitat i compromís social

8.1 Dimensió econòmica

En aquest projecte s'ha fet una avaluació dels costos dels recursos per la seva realització, tant materials com humans. Aquesta avaluació, visible en l'apartat anterior, té en compte tots els imprevistos de les tasques del projecte i plans d'actuació contra aquests.

La vida útil del projecte va més o menys lligada amb la vida útil de l'arquitectura RISC-V, com que és una tecnologia nova és difícil de preveure, però el recolzament[16] de grans empresses com: *Google*, *Samsung*, *Nvidia*, etc. Dona certa confiança a que aquest nou ISA s'acabi consolidant al mercat. Tot i així encara que l'arquitectura RISC-V quedes obsoleta per una altra arquitectura millor, aquest projecte encara serviria com a base per a l'assignatura de *PEC*.

La realització d'aquest projecte no comportarà cap millora econòmica a l'arquitectura existent a *PEC*, pot ser que fins i tot al contrari, s'hagi de fer una despesa econòmica per adquirir noves plaques per tal de suportar la nova arquitectura.

8.2 Dimensió ambiental

Durant el desenvolupament del projecte, no hi haurà cap element que comporti un gran impacte ambiental més enllà del consum energètic de l'ordinador i de la placa de desenvolupament els quals tenen un consum baix i és poden reaprofitar per a nous projectes.

Com s'ha discutit anteriorment en l'avaluació de l'estat de l'art, no és pot reaprofitar recursos d'altres projectes existents ja que las característiques d'aquests no s'adapten als objectius definits per aquest projecte.

Al ser RISC-V una arquitectura tan jove, la majoria d'implementacions d'aquesta no tenen com a objectiu principal reduir el consum del xip, tot i així, la pròpia arquitectura, amb un *instruction set* tan reduït, ja té millores ambientals (reducció de la mida del xip) respecte a les altres arquitectures del mercat. Aquest projecte no millorarà cap característica

existent (consum energètic o mida del xip) a la de altres projectes que treballen amb la mateixa arquitectura, n'hi en buscarà de noves.

8.3 Dimensió social

A nivell personal, aquest projecte m'aportarà una consolidació del objectius treballats a *PEC* així com també una bona familiarització de l'arquitectura RISC-V i el desenvolupament de hardware en FPGA.

Com s'ha esmentat anteriorment, en l'actualitat, les implementacions que trobem de l'arquitectura RISC-V són bastant complexes i suficientment difícils per alguna persona que no hagi treballat anteriorment en la industria. Es per això que al focalitzar el nivell de complexitat d'aquest projecte a un nivell educatiu, ajudarà a formar millors enginyers ja que permetrà una millor comprensió de l'arquitectura RISC-V i del disseny de computadors.

La necessitat d'aquest projecte, va lligada amb la necessitat d'actualitzar l'arquitectura amb la que es treballa a l'assignatura de *PEC*. Actualment l'arquitectura de *PEC* és l'arquitectura *SISA*, una arquitectura dissenyada a la *FIB* la qual també és utilitzada a l'assignatura d'*IC*. Aquesta arquitectura, tot i tenir gran part dels elements d'una arquitectura convencional, no és una arquitectura que es pugui trobar al mercat. És per això que una actualització de l'assignatura de *PEC* a una arquitectura RISC-V aportaria a més a més de tot el suport que hi ha per a RISC-V, treballar amb una arquitectura que després et podràs trobar al mercat.

Capítol 9

Conclusions

En aquest capítol, comentarem les conclusions que s'extreuen de la finalització del projecte, quins han estat els objectius assolits respecte als plantejats inicialment, quins aspectes han estat necessaris treballar amb més profunditat per assolir els objectius, la valoració final del processador i finalment quines possibles línies de futur es proposen per al projecte.

Dels objectius plantejats inicialment s'han aconseguit portar a terme satisfactòriament: implementar un processador multi-etapes que executa les instruccions definides en el RV32IM amb suport per a mode usuari, interrupcions i excepcions. Implementar un sistema amb l'eina *Platform Designer* que permeti controlar els diferents dispositius de la placa de desenvolupament, el qual ha requerit de força dedicació per formar-me i familiaritzar-me amb el *Platform Designer* i els diferents controladors utilitzats. L'únic objectiu no complert ha sigut la implementació de la extensió de coma flotant, el qual ja estava previst no fer-lo en cas de falta de temps.

Per tal de dur a terme el projecte, ha sigut necessari millorar el domini i coneixement en els diferents programes de la suite de RTL d'Intel, especialment el *Platform Designer*, i del llenguatge VHDL per tal de tenir una implementació concisa i ordenada. També ha sigut necessari familiaritzar-se amb les especificacions de l'ISA RISC-V, especialment amb els mòduls RV32I, M (mòdul de multiplicació i divisió d'enters) i la part on es descriuen els diferents nivells de privilegis, interrupcions i excepcions.

La valoració final del processador i del treball realitzat creiem que ha de ser positiva, ja que s'ha aconseguit una implementació clara i concisa del processador la qual serà fàcilment reproduïble per un alumne que cursi PEC. També l'ús i documentació de les diferents IPs aportarà noves possibilitats a l'assignatura de PEC i en millorarà el temps de certes tasques com ara la càrrega de programes a les memòries RAM de la placa de desenvolupament.

Finalment, presentarem algunes idees per a treballs futurs per tal de millorar la seva eficiència o ampliar les seves funcionalitats.
Primer de tot, per tal millorar el temps d'execució del processador sense modificar-ne el seu comportament, seria necessari afegir un nivell de catxe a la jerarquia de memòria, ja que com hem vist, els temps de lectura de la SDRAM són elevats. D'aquesta manera, aconseguiríem,

en cas de *hit*, temps de lectura d'un cicle.
Una segona millora al processador a fi d'aconseguir millors temps d'execució seria la segmentació del datapath, aprofitant les etapes ja implementades. Al segmentar el processador, també es podria afegir un predictor de salts estàtic, el qual no és molt complex d'implementar i ofereix certa millora de rendiment en les operacions de salt condicional.

Per tal d'afegir noves funcionalitats al processador, coherents amb el disseny que s'ha proposat, és podria implementar la unitat de coma flotant descrita en l'extensió F de l'ISA RISC-V o la unitat vectorial descrita en l'extensió V de l'ISA RISC-V. Una possible ampliació del processador, també podria ser la creació d'un sistema operatiu o fins i tot adaptar una versió del kernel de Linux, tot i que això comportaria molt més treball.

Bibliografia

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

[2] The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10", Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, May 2017.

[3] *Web amb el llistat de cores i SoC de RISC-V*, Recuperat de https://riscv.org/risc-v-cores/

[4] *Repositori del projecte Rocket*, Recuperat de https://github.com/freechipsproject/rocket-chip

[5] *Repositori dels processadors Riscy*, Recuperat de https://github.com/csail-csg/riscy

[6] *Web oficial del programa TeXstudio*, Recuperat de https://www.texstudio.org/

[7] *Web oficial de l'aplicació Gantter*, Recuperat de https://www.gantter.com/

[8] Patterson, David, and Andrew Waterman. Guía Práctica de RISC-V: El Atlas de una Arquitectura Abierta. Trans. Alí Lemus i Eduardo Corpeño. 1.0.5 ed. PDF.

[9] Patterson, David, and John L. Hennessy. Computer organization and design risc-v edition. Boston, MA: Elsevier, 2018.

[10] *RARS, programa de simul·lació de l'arquitectura RISC-V*, Recuperat de https://github.com/TheThirdOne/rars

[11] *Web oficial del programa*, Recuperat de https://www.intel.com/content/www/us/en/programmable/downloads/download-center.html

[12] *Web oficial del programa*, Recuperat de https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html

[13] *Binutils de l'arquitectura RISC-V*, Recuperat de https://github.com/riscv/riscv-tools

[14] *Pagina web oficial del programa Git*, Recuperat de https://git-scm.com/

[15] *Pagina web oficial de la placa de desenvolupament DE2-115*, Recuperat de https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=502

[16] *Web oficials dels membres de la RISC-V Foundation*, Recuperat de https://riscv.org/members-at-a-glance/

[17] *Estudio de remuneración Michael Page*, Recuperat de http://www.michaelpage.es/sites/michaelpage.es/files/ingenieros2016.pdf

[18] *Preu kWh*, Recuperat de https://comparadorluz.com/tarifas/precio-kwh

[19] *Documentació del SoC FU540-C000*, Recuperat de https://sifive.cdn.prismic.io/sifive%2F834354f0-08e6-423c-bf1f-0cb58ef14061_fu540-c000-v1.0.pdf

[20] *Documentació dels IP Cores del Platform Designer*, Recuperat de https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_embedded_ip.pdf

[21] *Documentació de la interfície Avalon*, Recuperat de https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf

Apèndix A

Taula i Diagrama de Gantt de la planificació

Nombre	Duración	Inicio	Fin	Predecesoras
Familiarització amb l'ISA RISC-V i l'arquitectura dels processadors	5días	17/01/2019	23/01/2019	
Gestió del projecte	**21días**	**20/02/2019**	**20/03/2019**	
Abast del projecte i contextualització	5días	20/02/2019	26/02/2019	1
Planificació temporal	2días	01/03/2019	04/03/2019	3
Gestió econòmica i sostenibilitat	2días	08/03/2019	11/03/2019	4
Presentació del projecte i document final	7días	12/03/2019	20/03/2019	5
Anàlisi del projecte	2días	24/01/2019	25/01/2019	1
Entorn de treball i placa de desenvolupament	19días	29/01/2019	22/02/2019	7
Disseny i implementació	**73.13días**	**18/02/2019**	**30/05/2019**	
Processador monocicle	4.13días	18/02/2019	22/02/2019	7
Processador mulit-clicle	10.13días	25/02/2019	11/03/2019	10
Memòria	11días	11/03/2019	26/03/2019	11
ALU	2días	15/03/2019	18/03/2019	11
Salts	4días	19/03/2019	22/03/2019	13
Entrada/Sortida	2días	26/03/2019	28/03/2019	12,14
PS2 i VGA	8días	28/03/2019	09/04/2019	15
Interrupcions i excepcions	20días	09/04/2019	07/05/2019	16
Mode sistema	12días	03/05/2019	20/05/2019	14
Operacions de coma flotant	7días	22/05/2019	30/05/2019	11
Memòria final	11días	03/06/2019	17/06/2019	17,18,19

Figura A.1: Taula de Gantt amb les dates de inici i fi de les tasques

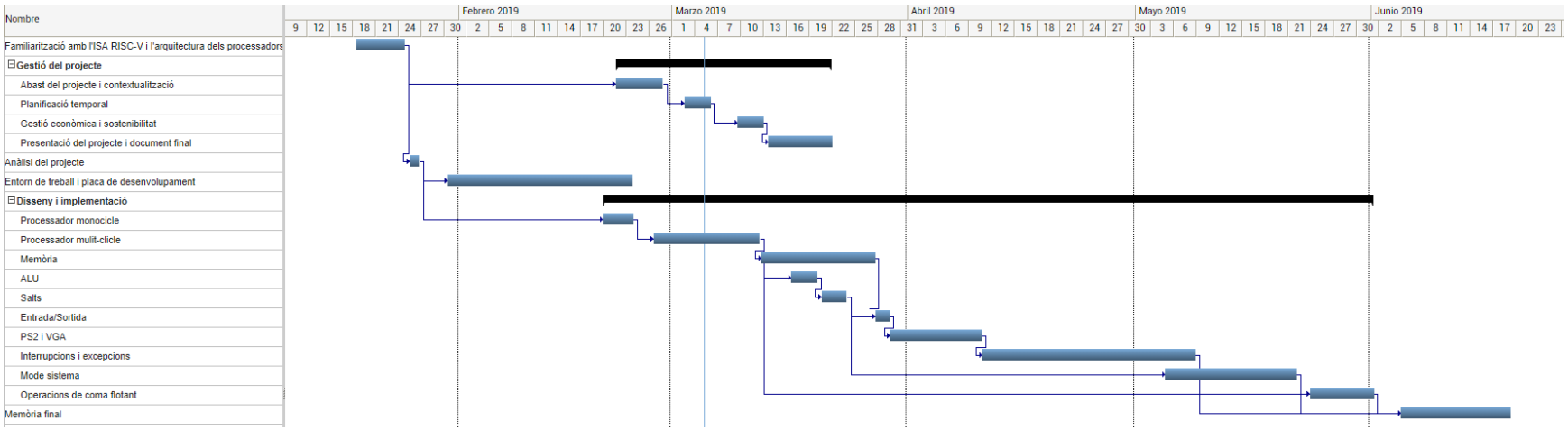


Figura A.2: Diagrama de Gantt amb la progressió temporal de les tasques

Apèndix B

Descripció del RV32IM

La versió de l'ISA RISC-V que ha implementat aquest projecte ha sigut la *RV32IM*, es a dir, un processador de 32 bits, el qual implementa el mòdul estàndard d'operacions amb enters i el mòdul M, on estan definides les operacions de multiplicació i divisió d'enters. Aquesta implementació també suporta dos dels tres modes de funcionament descrits en el document *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*, el mode usuari i el mode sistema.

El processador compta amb 32 registres de 32 bits, d'ús general per al programador, dels quals un, el *zero*, conté sempre el valor zero i qualsevol escriptura en aquest registre queda descartada. Pel que fa als registres de sistema, anomenats CSR, l'ISA en defineix 4096 tot i que la majoria estan reservats per a futures extensions. Per tal d'accedir a aquests registres, es necessari utilitzar unes instruccions especials les quals estan protegides si el processador està funcionant en mode usuari.

Per tal de tractar interrupcions, l'ISA descriu el comportament d'un core addicional anomenat PLIC (*Platform Level Interrupt Controller*), aquest core o dispositiu, mapejat a memòria com tots els altres dispositius, s'encarrega de gestionar totes les interrupcions externes al core RV32IM i en el cas d'haver-hi més d'un core, també és l'encarregat d'enviar la interrupció al core corresponent.

L'ISA RISC-V, no defineix l'organització de l'espai d'adreces del processador la qual deixa en mans dels dissenyadors, així com també l'adreça inicial del processador o de les rutines d'interrupcions o excepcions. Pel que fa a la memòria virtual, l'ISA defineix el comportament de certs mòduls opcionals, amb els seus CSRs, encarregats de la traducció de les adreces, gestió de les pàgines de memòria i la protecció d'aquestes. Tot i així l'ús de memòria virtual és opcional, i serà el dissenyador l'encarregat de decidir si s'implementa o no.

Per tal d'obtenir encara més informació sobre l'ISA RISC-V es recomana una lectura ràpida del llibre *Guía Práctica de RISC-V: El Atlas de una Arquitectura Abierta*[8] el qual ofereix una visió informativa dels aspectes més importants de l'arquitectura.

Apèndix C

Eines per al desenvolupament de codi

Per tal de compilar i simular codi assembler risc-v, s'ha utilitzat l'eina RARS[10]. Un simulador fet en JAVA el qual permet la compilació i execució pas a pas d'un codi programat amb l'assembler de RISC-V. A més a més, també permet extreure la codificació hexadecimal de les instruccions i dels resultats del programa guardats a memòria. Poder obtenir aquests valors hexadecimals, ens anirà bé per després carregar el programa a la memòria SDRAM i verificar-ne els resultats.

Una altre eina que podem utilitzar per compilar l'assembler risc-v o codi C, són les *binutils* de gcc[13]. Una vegada compilat el programa i haver obtingut el binari, amb les eines *as* o *gcc*, assembler o codi C respectivament, podem utilitzar l'eina *objdump* sobre el binari per tal d'obtenir els codis hexadecimals de les instruccions.

Finalment, una de les avantatges que ha comportat l'ús dels IP cores, ha sigut la possibilitat de poder gestionar la memòria SDRAM, i el altres dispositius, mitjançant la connexió JTAG. Això comporta que mitjançant l'eina *System Console*, que suporta mode CLI (*Command-Line Interface*), puguem escriure i llegir el contingut de la SDRAM. Per tal de facilitar la feina de carregar un programa a la SDRAM i comprovar-ne el resultat una vegada executat, s'ha creat un petit programa amb Python. Aquest programa, s'encarrega de generar un binari que conté les instruccions, crear els scripts TCL per carregar el binari a la SDRAM mitjançant el *System Console*, llegir els resultats de la SDRAM i comprovar si són correctes.

Tots aquests *scripts*, es poden trobar als arxius adjunts a aquesta memòria, així com també al repositori de GitHub `https://github.com/Sustrak/RISC-V`.