Títol: Implementació d'un processador MIPS en una FPGA

Volum: 1/1

Alumne: David Guillen Fandos

Director/Ponent: José M. Llabería Griñó

Departament: Arquitectura de Computadors

Data: 18/06/2012

DADES DEL PROJECTE

Títol del Projecte: Implementació d'un processador MIPS en una FPGA

Nom de l'estudiant: David Guillen Fandos
Titulació: Enginyeria Informàtica
Crèdits: 37.5
Director/Ponent: José M. Llabería Griñó
Departament: Arquitectura de Computadors

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President:

Vocal:

Secretari:

QUALIFICACIÓ

Qualificació numèrica:
Qualificació descriptiva:

Data:

Índex

Annexes

Capítol 1

Objectius

La indústria dels circuits integrats ha patit una gran evolució en les darreres dècades. Mentre al l'inici dels anys setanta es parlava de dissenys amb milers de transistors a data d'avui es parla de xifres milions de vegades superiors. Aquest progrés en el desenvolupament de components circuitals ha provocat l'aparició de solucions molt diverses que responen a les necessitat de la indústria. Una solució en particular són les *FPGA*.

Les *FPGA*s són uns dispositius electrònics programables que permeten sintetitzar circuits amb característiques similars als fabricats amb procediments tradicionals però més accessibles als usuaris no professionals. La seva facilitat d'ús i el baix preu (en comparació als processos tradicionals de fabricació) han promogut el seu ús en diverses àrees, especialment en el disseny de circuits integrats per aplicacions específiques (*ASIC*).

En particular, la síntesi de processadors en *FPGA*s ha resultat ser de gran utilitat per a la implementació de solucions especifiques. Les solucions basades en l'ús de processadors i microcontroladors aporten totes les bondats de l'enginyeria del software al món del disseny digital: dissenys fàcils de mantenir i ampliar a la vegada que de cost més reduït.

1. Objectius del projecte

L'objectiu del projecte és el disseny i implementació d'un computador al voltant d'un processador *MIPS*. Aquest computador ha de funcionar a una placa de demostració *Terasic DE2-115*, que disposa d'una *FPGA Altera Cyclone IV*. Dividirem el treball en els següents tres punts principals.

1.1 Disseny d'un processador *MIPS*

Es desitja dissenyar un processador que implementi l'arquitectura *MIPS*. S'implementarà una versió senzilla del *MIPS* que, tot i no suportar totes les instruccions existents, donarà lloc a un processador amb totes les funcionalitats. El processador implementarà una arquitectura *Harvard*, és a dir, amb memòries d'instruccions i dades separades. Serà capaç de treballar en dos modes de funcionament: un mode privilegiat i un mode no privilegiat (també anomenat d'usuari).

Es desitja assolir aquests objectius de forma eficient i per aquest motiu el processador es dissenyarà per a que sigui segmentat. A més es farà ús de tècniques per a reduir el nombre de cicles perduts en interpretar instruccions i aconseguir una taxa de cicles per instrucció propera a la unitat.

La implementació es realitzarà en el llenguatge *VHDL* sobre l'entorn de desenvolupament proporcionat pel fabricant Altera. Caldrà adaptar el disseny a les restriccions imposades pel hardware.

1.2 Implementació de dispositius d'entrada-sortida

Amb l'objectiu d'utilitzar el computador s'implementaran dispositius que permetin la interacció d'aquest amb l'usuari. Aquests dispositius seran en particular el teclat i la pantalla. D'aquesta manera es farà ús d'alguns dels dispositius que proporciona la placa.

La placa disposa d'una interfície *VGA* per a la implementació de la pantalla i d'un bus *PS/2* per al teclat. Els objectius seran aprendre el funcionament d'aquests i implementar dispositius que puguin ser utilitzats per l'usuari.

1.3 Implementació d'un nucli de sistema

Per a fer ús dels dispositius s'implementarà un petit nucli de sistema operatiu que en gestioni l'accés. Es desitja que aquest nucli realitzi les operacions bàsiques d'un sistema operatiu. En primer lloc implementarà les crides a sistema per part dels programes d'usuari així com les excepcions que aquests puguin produir. Realitzarà gestió de processos d'usuari i finalment atendrà les interrupcions que puguin generar els dispositius d'entrada.

Es desitjable que el nucli segueixi les especificacions *POSIX* o s'hi apropi en la mesura del possible. Es prendrà com a referència el sistema operatiu Linux en la seva implementació per a *MIPS*.

Capítol 2

Introducció

Aquest capítol fa una descripció del context del projecte. En primer lloc s'introduirà la família de hardware sobre la que es desenvoluparà el projecte. Tot seguit es parlarà del llenguatge utilitzat per a especificar-lo. Per últim es descriurà la metodologia de desenvolupament emprada en aquest tipus de projectes.

1. Arquitectures hardware programables

Els dispositius lògics programables són un tipus de circuits digitals que permeten configurar, amb un cert grau de flexibilitat, la seva estructura interna per a implementar diversos circuits. Estan formats per conjunts de blocs de portes que poden ser connectades entre sí a voluntat del dissenyador amb unes certes restriccions.

Aquests dispositius presenten una gran varietat de famílies i arquitectures ja que responen a una definició molt oberta. De fet, es consideren dispositius d'aquest tipus les memòries programables (*PROMS*, *EEPROMs*) perquè permeten crear lògica combinacional i configurar la seva estructura amb molta llibertat.

1.1 Història de les principals arquitectures

Els primers dispositius dedicats a aquesta tasca que varen sortir al mercat van ser les *PLA* (*Programable Logic Array*), que són uns components programables una sola vegada, els quals permeten implementar funcions lògiques fent ús de AND/OR (minterms, o suma de productes). El dispositiu està format per portes arranjades de forma canònica (minterms) i amb totes les possibles connexions [1]. Per a configurar-ne la seva estructura interna es cremen uns petits fusibles per a trencar les connexions no necessàries. A la *figura 2.1* es pot veure un exemple de PLA amb dos ports d'entrada i un de sortida que permet implementar qualsevol funció.

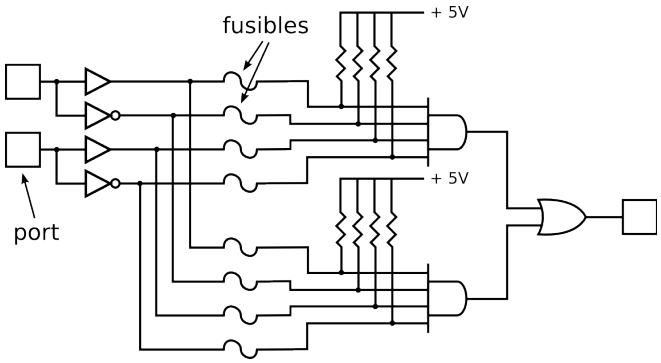


Figura 2.1: Exemple de *PLA* amb dues entrades i una sortida

Més endavant van aparèixer les *GAL*s (*Generic Array Logic*), uns dispositius que tenen la propietat de ser reprogramables. Aquesta capacitat de re-programació es va aconseguir fent ús de la mateixa tecnologia que les *EEPROM*s. L'evolució d'aquestes van ser les *CPLD*s (*Complex Programmable Logic Device*) que van aportar millores substancials permetent funcions complexes (no només de forma canònica) tals com operacions aritmètiques, unitats de memòria, etc. [2]

A partir d'aquest punt es van començar a estructurar els blocs interns en blocs bàsics (també anomenats pels acrònims anglosaxons *LE*s o *LUT*s), que són blocs que implementen una unitat lògica mínima. Típicament contenen un o més registres i un bloc combinacional petit (d'unes 4 entrades). La utilització de blocs de lògica més complexe (com ara multiplexors o sumadors) provoca una millora de les prestacions (en temps de cicle i nombre de portes utilitzades), ja que permet fer ús de blocs que, de no existir, caldria implementar fent ús de blocs bàsics. A la *figura 2.2* es mostra un esquema de com s'interconnecten aquests blocs bàsics. Fan servir xarxes de busos (matriu d'interconnexió a la figura) que es poden connectar a voluntat. A la *figura 2.3* es mosta un bloc bàsic d'una *CPLD* format per dos blocs combinacionals (*3-LUT*), un sumador (*FA*), multiplexors i un registre d'un bit.

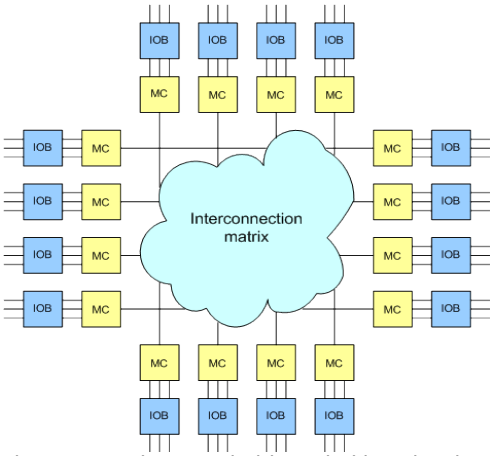


Figura 2.2: Digrama de blocs de l'interior d'una *CPLD*

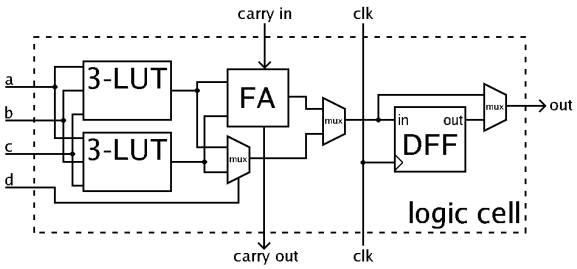


Figura 2.3: Diagrama de blocs d'una *Logical Entity* comercial

Actualment han aparegut les *FPGA*s (*Field Programable Gate Array*) que contenen gran quantitat de funcions específiques que es fan servir típicament en el disseny digital. Aquestes millores inclouen *DSP*s (*Digital Signal Processor*), multiplicadors, memòria dedicada, registres, *DAC*s (*Ditigal-Analog Converter*) i *ADC*s (*Analog-Digital Converter*), etc. A més els blocs bàsics s'han especialitzat una mica més, per exemple en lògica combinacional i seqüencial.

1.2 Aplicacions

Les arquitectures de hardware programables s'utilitzen típicament en aplicacions molt específiques. Normalment es tria aquestes arquitectures enlloc d'un microcontrolador quan aquest no pot realitzar les tasques necessàries o resulta més complexe i car [3]. Com a exemple es pot citar solucions de processat de senyal (comunicacions, codificació i encriptació) on són molt útils ja que permeten sintetitzar dissenys que explotin la capacitat de processament en paral·lel, així com controladors per a dispositius específics (monitors, aplicacions industrials, etc.).

L'alt grau d'especialització dels dispositius que es poden implementar provoca moltes vegades hardware difícil de dissenyar i car de mantenir (en comparació a hardware equivalent basat en microcontroladors). Per aquest motiu és habitual trobar una combinació de microcontrolador i una *FPGA* en sistemes comercials. Per a evitar el sobre cost que això suposa es pot optar per solucions mixtes. Una solució habitual pot ser la síntesi d'un microcontrolador a la FPGA, d'aquesta manera es tenen els avantatges de les dues arquitectures en un sol component.

Des de fa un temps diferents fabricants de components proporcionen implementacions de diferents processadors per a un ràpid desenvolupament de projectes [4]. Aquests processadors (anomenats de vegades *soft-core*) tenen característiques molt interessants ja que permeten una certa flexibilitat en la seva síntesi: nombre d'etapes de la segmentació, mida de la memòria *cache*, etc...

2. Llenguatges de descripció de hardware

Els llenguatges de descripció de hardware (*HDL*, de l'anglès *Hardware Description Languages*) són, com el seu nom indica, llenguatges formals de programació que permeten descriure circuits electrònics. El seu objectiu és detallar el funcionament de circuits amb precisió i sense ambigüitat per tal de poder-ne realitzar simulacions i posteriorment sintetitzar-los.

2.1 Història i llenguatges estandarditzats

La seva història es remunta al voltant de l'any 1977 quan aparegueren els que serien els primers llenguatges: *ISPS* (*Instruction Set Processor Specification*) i *KARL* (*Kaiserslautern Register Transfer Language*). Aquests van ser desenvolupats en resposta a la necessitat d'especificar i simular circuits digitals i, per tant, no permetien implementar (altrament anomenat sintetitzar) circuits, ja que es limitaven a descriure relacions entre entrades i sortides [5].

Més endavant van sorgir llenguatges amb més funcionalitats, desenvolupats a universitats o empreses, amb l'objectiu de cobrir necessitats particulars d'aquestes. Un llenguatge en concret anomenat *Verilog* va destacar d'entre tots per aconseguir una gran adopció en la indústria. Patrocinat per una empresa gran en el sector com és *Cadence*, va arribar a ser un estàndard *de facto* [6].

Tot i això al 1987 el departament de defensa dels EUA va patrocinar la creació d'un llenguatge més robust i adequat per a grans dissenys que es va anomenar *VHDL*. Degut a l'estandardització d'aquest per part del *IEEE* (1076-1987 i 1076-1993) va passar a ser un llenguatge molt utilitzat, especialment en l'àmbit acadèmic [7].

Finalment *Verilog* va aconseguir també l'estandardització per part del *IEEE* (1364-2001) i d'aquesta manera *Verilog* i *VHDL* van passar a ser els dos llenguatges estàndards més utilitzats per a la descripció de circuits electrònics fins l'actualitat.

2.2 El llenguatge VHDL

El modelat de circuits amb VHDL es realitza fent ús d'entitats i arquitectures. Una entitat és la descripció externa d'un bloc circuital, és a dir, una descripció del circuit com si aquest fos una caixa negra. Només especifica les entrades i sortides però no descriu què hi ha a dins. Una arquitectura, en canvi, és la descripció interna d'un bloc circuital. D'aquesta manera un bloc circuital queda completament definit quan s'especifica una entitat i una arquitectura. Com a detall cal dir que un bloc circuital només té una entitat però pot tenir diferents arquitectures.

2.2.1 Entitats

Les entitats consisteixen en un conjunt de senyals d'entrada i de sortida. Cada senyal s'identifica amb un nom i un sentit (entrada o sortida) i té, a més, un tipus associat. Per exemple es pot tenir una senyal binària (un cable, que porta un bit) o bé un conjunt de senyals binàries (un bus, que porta més d'un bit).

En l'exemple de la *figura 2.4* es pot veure una entitat que té dues entrades i una sortida. L'entrada *x* és una senyal binària mentre que la senyal *y* és un bus de 3

bits. La sortida *z* és un bus de 2 bits. La seva descripció en VHDL està exemplificada a la *figura 2.5*.

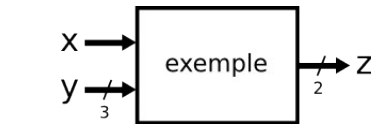


Figura 2.4: Representació d'una entitat

```
entity exemple is
  port (
    x : in  std_logic;
    y : in  std_logic_vector(2 downto 1);
    z : out std_logic_vector(1 downto 0)
  );
end entity
```

Figura 2.5: Definició VHDL de l'entitat “exemple”

2.2.2 Arquitectures

Les arquitectures descriuen la relació entre les entrades i les sortides. Això es pot fer de tres maneres diferents anomenades: arquitectura estructural, arquitectura de flux de dades (*dataflow*) i arquitectura de comportament (*behavioural*).

L'arquitectura estructural descriu el funcionament d'un bloc circuital a partir d'altres blocs circuitals interconnectats entre sí. A la *figura 2.6* es mostra un exemple on s'implementa una porta *xor* fent ús de portes *and*, *or* i *not*.

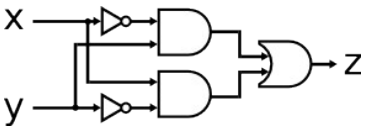


Figura 2.6: Implementació d'una porta *xor* fent ús d'altres portes.

```
architecture estructural of portaxor is
  signal notx,noty,s1,s2 std_logic;

  component notgate
    PORT (a: in std_logic, b: out stdlogic);
  end component;

  component andgate
    PORT (a,b: in std_logic, c: out stdlogic);
  end component;

  component orgate
    PORT (a,b: in std_logic, c: out stdlogic);
  end component;

begin
  not1 : notgate PORT MAP (x,notx);
  not2 : notgate PORT MAP (y,noty);
  and1 : andgate PORT MAP (y,notx,s1);
  and2 : andgate PORT MAP (x,noty,s2);
  or1  : orgate  PORT MAP (s1,s2,z);
end architecture
```

Figura 2.7: Arquitectura estructural de l'entitat *portaxor*

El llistat de la *figura 2.7* mostra com es fa la descripció en llenguatge *VHDL*. Primer s'especifiquen els blocs que es faran servir (anomenats components) descrivint-ne les seves entitats. Després es creen instàncies d'aquests components i, fent ús de *PORT MAP*, s'especifiquen les connexions entre blocs. Per a realitzar connexions internes (que no siguin només amb ports d'entrada o sortida) es fan servir *SIGNAL*s, que són cables interns del bloc.

L'arquitectura de *dataflow* es basa en el principi de la interconnexió de senyals. Fent ús d'operadors (*and*, *not*, etc.) i sentències pròpies del llenguatge especifica les

sortides en funció de les entrades. A la *figura 2.8* es mostra un exemple d'una porta *xor* (de la *figura 2.6*) implementada fent ús de *dataflow*.

```
architecture dataflow of portaxor is
begin
  z <= (not x and y) or (not y and x);
end architecture
```

Figura 2.8: Arquitectura *dataflow* de l'entitat *portaxor*

L'arquitectura de comportament descriu el circuit a molt alt nivell, com si fos un programa d'ordinador. Apareixen els anomenats "processos", que són blocs de codi que s'executen sempre que es produeix una variació en certes senyals de l'anomenada "llista de sensibilitat". La *figura 2.9* mostra una implementació d'una porta *xor* fent ús d'arquitectura de comportament. Aquesta arquitectura està especialitzada en la descripció de lògica seqüencial de forma senzilla. A la *figura 2.10* es mostra un exemple d'implementació d'un registre per flanc de pujada.

```
architecture comportament of portaxor is
begin
  process(x,y)
  begin
    z <= (not x and y) or (not y and x);
  end process;
end architecture
```

Figura 2.9: Arquitectura *dataflow* de l'entitat *portaxor*

```
architecture comportament of regD is
begin
  process(D,clk)
  begin
    if (clk'event and clk='1') then
      Q <= D;
    end if;
  end process;
end architecture
```

Figura 2.10: Arquitectura *dataflow* de l'entitat *portaxor*

3. Disseny de sistemes digitals

El procés de disseny d'un sistema digital es pot descriure, si es fa ús de llenguatges de descripció de hardware, en quatre etapes: especificació, disseny, implementació i testeig. Aquestes quatre etapes es poden repetir de forma cíclica durant el desenvolupament i/o el temps de vida del sistema (*figura 2.11*).

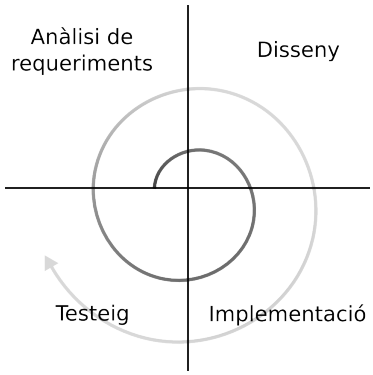


Figura 2.11: Cicle de vida dels dissenys de circuits digitals

Les etapes d'implementació i testeig són, en realitat, dues etapes d'implementació i dues de comprovació degut a la naturalesa pròpia del sistemes hardware [8]. Les dues primeres etapes consisteixen en una implementació a nivell lògic i una

simulació realitzada per software mentre que les segones, en canvi, generen una implementació física que és comprovada físicament (*figura 2.12*).

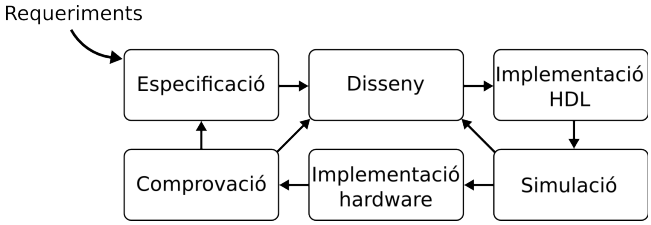


Figura 2.12: Diagrama de les etapes de disseny d'un sistema digital

La necessitat de dissenyar i simular el circuit abans de fer-ne una síntesi sorgeix de l'elevada complexitat dels circuits. Resulta molt difícil trobar i corregir errors en un circuit una vegada ha estat sintetitzat. Per altra banda el cost econòmic que suposa la simulació és molt inferior al de la fabricació d'un circuit real.

3.1 Implementació i testeig lògic

L'etapa de disseny lògic es desenvolupa fent ús d'un llenguatge de descripció de hardware. En el cas particular de *VHDL* es desenvolupen blocs circuitals fent ús d'entitats i les seves corresponents arquitectures. Cada una de les entitats desenvolupades passa un procés de testeig en el qual es comprova el seu funcionament. Aquest procés consisteix en jocs de prova, que són parelles entrada-sortida de valors precalculats mitjançant els quals s'assegura que el sistema respon correctament. Un exemple de joc de prova (també anomenat *testbench*) en *VHDL* es pot veure a la *figura 2.13*.

```
architecture arch of testbench is
  signal ina,inb,rout,eout: std_logic;
begin
  elem: portaxor PORTMAP (ina,inb,rout);
  process
  begin
    ina <= '0'; inb <= '0'; eout <= '0'; wait for 20ns;
    ina <= '1'; inb <= '0'; eout <= '1'; wait for 20ns;
    ina <= '0'; inb <= '1'; eout <= '1'; wait for 20ns;
    ina <= '1'; inb <= '1'; eout <= '0'; wait for 20ns;
    wait;
  end process;
end architecture
```

Figura 2.13: Arquitectura d'un joc de proves per l'entitat *portaxor*

Les simulacions es realitzen a nivell lògic, és a dir, de forma totalment independent de la tecnologia. Tot i això és possible realitzar simulacions del circuit final (el que es sintetitzarà) fent ús de software específic. Aquestes simulacions són molt útils principalment per a determinar les característiques finals del circuit tals com el nombre de transistors que farà servir així com el retard, el temps de cicle, els camins crítics, etc. Depenent dels requeriments una simulació a nivell de portes pot ser imprescindible per a garantir que es satisfan els requeriments.

3.2 Síntesi

L'etapa de síntesi és la implementació real del circuit. És en aquesta etapa on el circuit descrit en llenguatge d'alt nivell es tradueix a en components concrets com ara portes lògiques, transistors, etc. Abans de l'aparició dels *HDL*, aquesta etapa es realitzava de forma manual: el dissenyador especificava portes i/o transistors i les seves connexions.

L'aparició dels *HDL* com a eines de simulació va evolucionar fins a convertir-los també en eines de síntesi, permetent la síntesi de circuits a partir de la seva descripció en *HDL*. Aquest procés de síntesi automàtica el realitza un compilador que s'encarrega de produir dissenys de baix nivell a partir del codi. La gran varietat de dispositius programables existents provoca que existeixin diferents compiladors depenent del hardware on s'implementarà el circuit.

A la *figura 2.14* es pot veure la disposició (*routing*) i ús dels diferents blocs d'una *FPGA* per a un circuit d'exemple. Els blocs bàsics s'interconnecten entre sí a través dels busos presents a la *FPGA*. A la *figura 2.15* es pot veure un disseny d'un multiplexor implementat fent ús de transistors. Aquests dissenys consisteixen en especificar transistors (mida, posició, etc.) i les seves connexions. Es pot apreciar la gran diferència entre arquitectures tot i partir d'un disseny comú especificat en un llenguatge d'alt nivell.

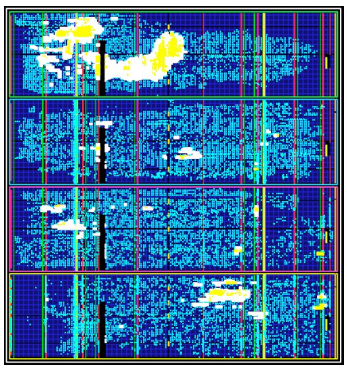


Figura 2.14: Exemple d'enrutat d'un circuit a una *FPGA*

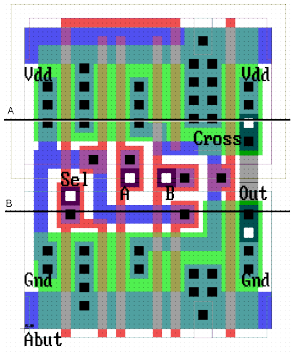


Figura 2.15: Exemple de *layout* de transistors

L'heterogeneïtat dels dispositius provoca que la síntesi d'un mateix codi es realitzi de forma molt diferent en cada un d'aquests. Els fabricants de hardware i els dissenyadors dels compiladors permeten una certa flexibilitat en les síntesi de dissenys, però no es poden sintetitzar tots els programes *VHDL* possibles. Per aquest motiu els fabricants recomanen certes metodologies d'implementació amb *VHDL* per tal de poder assegurar que el disseny serà sintetitzable.

Capítol 3

Descripció del processador

Per a descriure un processador distingirem per una banda l'anomenada *ISA* (*Instruction Set Architecture*) i per altra banda la microarquitectura.
La *ISA* és l'especificació que descriu quines instruccions és capaç d'interpretar el processador. Aquesta especificació és utilitzada per a crear programes per al processador, ja que descriu amb detall la codificació de les instruccions així com la seva interpretació.
La microarquitectura és la implementació de la *ISA*. Com a tal, no s'exposa al programador i només és rellevant per als dissenyadors i fabricants del processador. Donada una *ISA* poden existir nombroses microarquitectures que l'implementin.
Es començarà per donar una breu descripció de la *ISA* del processador *MIPS* i es descriurà una implementació segmentada d'aquesta.

1. Llenguatge màquina

El llenguatge de l'arquitectura MIPS està format per instruccions de mida fixa de 32 bits. Aquestes instruccions es poden dividir en tres grans grups: operacions entre registres, operacions amb un literal i operacions de salt. El format de cada un d'aquests grups s'explica a continuació. Totes les instruccions comencen per un codi d'operació de 6 bits que indica a quin grup pertany la instrucció així com quina operació realitza [9].

Les instruccions entre registres, anomenades també instruccions *tipus R*, realitzen operacions fent ús dels valors dels registres (*figura 3.1*). Els camps *RS*, *RT* i *RD* (*figura 3.3*) son nombres de 5 bits que representen un registre del banc de registres. Aquests camps tenen com a finalitat indicar quin parell de registres es fan servir per a operar (*RS*, *RT*) i a quin registre es guarda el resultat de l'operació (*RD*). Addicionalment es troba un segon codi d'operació de sis bits anomenat *FUNC* i un petit literal de cinc bits anomenat *SA* (*figura 3.3*).

RD = *RS* op *RT*	Registre-registre
RD = *RT* op *SA*	Registre-literal

Figura 3.1: *Operacions que pot realitzar una instrucció tipus R*

Les instruccions amb un literal (o de *tipus I*) fan ús d'una parella de registres i d'un literal codificat a la mateixa instrucció. Les operacions que realitzen tenen com operands un dels dos registres (*RS*) i un literal, codificat al camp *IMM* (*figura 3.4*). A la *figura 3.2* es mostra un resum de les operacions que poden realitzar.

RT = *RS* op *IMM*	Registre-literal
RT = MEM[*RS* + IMM]	Lectura memòria
MEM[*RS* + IMM] = *RT*	Escriptura memòria
Si *cond(RS,RT)* *PC* = *PC* + *IMM* + 4	Salt condicional

Figura 3.2: Operacions que pot realitzar una instrucció *tipus I*

Les instruccions de salt (o de *tipus J*) fan ús del literal de 26 bits codificat al camp *TARGET* (*figura 3.5*) per a canviar el comptador de programa.

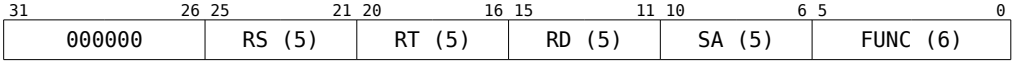


Figura 3.3: Codificació de les instruccions tipus R (*R-type*)

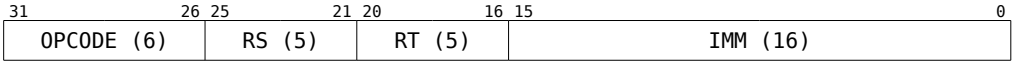


Figura 3.4: Codificació de les instruccions tipus I (*I-type*)

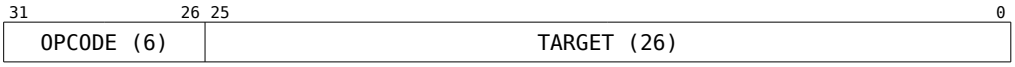


Figura 3.5: Codificació de les instruccions tipus J (*J-type*)

A més d'aquests tres grans grups també hi han altres operacions relacionades amb els anomenats coprocessadors. Aquestes instruccions tenen un codi d'operació de la forma 0100XX, on XX representa el número del coprocessador (dels 4 possibles) i no tenen una forma regular, ja que depenen en gran mesura del sistema (memòria, dispositius, etc.).

El processador treballa amb adreces de memòria de 32 bits, pel que és capaç d'adreçar 4 GB de memòria (amb granularitat d'un byte). Disposa també de 32 registres de 32 bits dels quals un d'ells (el número zero) sempre val zero.

2. Microarquitectura

Es descriu a continuació la implementació d'una microarquitectura segmentada lineal que interpreta el llenguatge màquina del processador *MIPS*. Aquesta està segmentada en sis etapes on cada una d'elles interpreta una part d'una instrucció. Les etapes s'executen de forma lineal començant per *CP* fins a *ES* (*figura 3.6*). A continuació es descriu breument cada una d'elles.

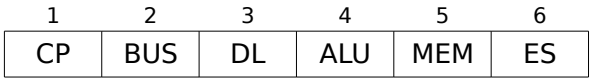


Figura 3.6: Etapes del processador *MIPS* segmentat

1. *CP*: Es determina la propera instrucció a ser cercada a memòria.

2. *BUS*: S'accedeix a la memòria d'instruccions per a llegir la instrucció indicada pel comptador de programa.

3. *DL*: Es descodifica la instrucció en un conjunt de senyals que indiquen a les etapes posteriors com processar la instrucció. També es realitzen lectures al banc de registres.

4. *ALU*: Es realitza un càlcul aritmètic o lògic. Es realitza també l'avaluació de la condició de salt i el càlcul de l'adreça de destí.

5. *MEM*: S'accedeix a memòria de dades per a llegir o escriure.

6. *ES*: S'escriuen els resultats d'etapes anteriors al banc de registres.

3. Camí de dades

Es pot diferenciar entre tres grans grups d'instruccions que fan un ús diferenciat del recursos del processador: instruccions aritmeticològiques, accessos a memòria i instruccions de salt. Totes les instruccions però comparteixen la utilització de recursos en les etapes *CP*, *BUS* i *DL*. En aquestes etapes es determina la propera instrucció (*CP*) i es llegeix la instrucció de la memòria d'instruccions (*BUS*). Després es descodifica la instrucció i es llegeixen els registres del banc de registres (*DL*).

3.1 Instruccions aritmeticològiques

Els dos operands, que són entrades de l'*ALU*, operen en aquesta segons indica el codi d'operació especificat pel descodificador. El resultat, que és la sortida de l'*ALU*, passa a través de l'etapa *MEM* (on no es fa res) fins arribar a l'escriptura al banc de registres a l'etapa *ES*.

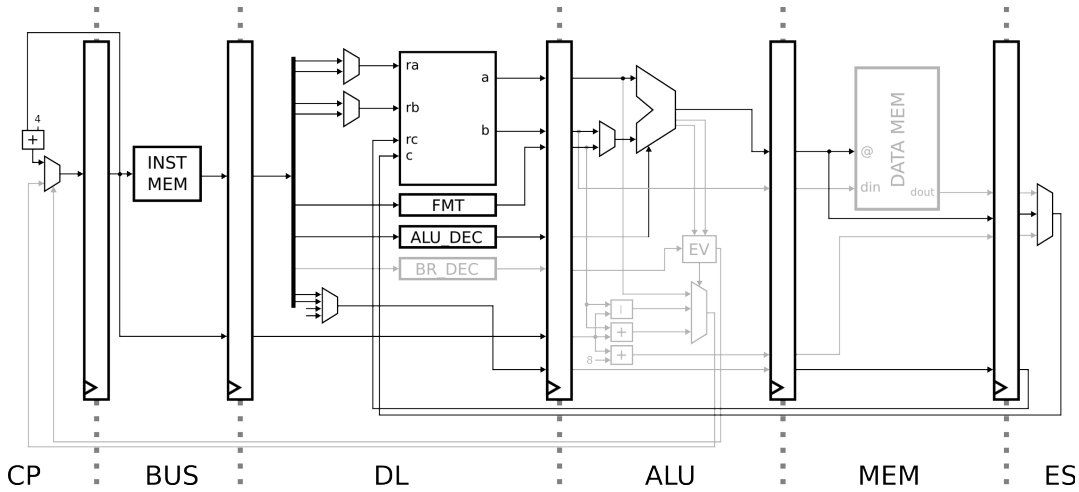


Figura 3.7: Ús del camí de dades per part de les instruccions que realitzen operacions aritmeticològiques

3.2 Instruccions d'accés a memòria

Es suma l'adreça base i el desplaçament a l'*ALU* per a obtenir l'adreça amb la qual s'accedirà a memòria. Si es tracta d'una lectura es propaga el resultat llegit a memòria fins a *ES* per a la seva escriptura al banc de registres. En cas d'una escriptura a memòria es fa ús del valor d'un registre com a dada a guardar.

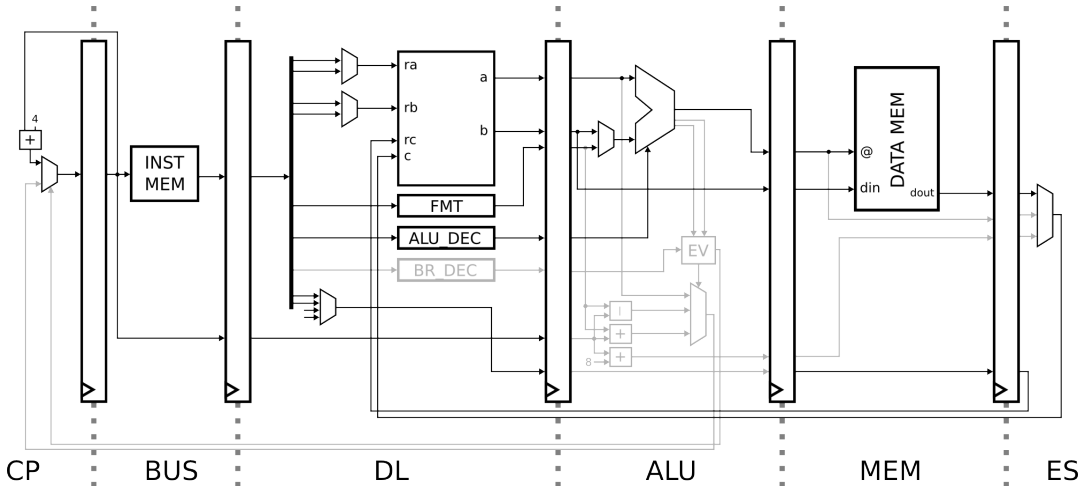


Figura 3.8: Ús del camí de dades de les instruccions d'accés a memòria

3.3 Instruccions de salt

Es distingeixen instruccions de salt i instruccions de crida a subrutines. Les primeres canvien el comptador de programa i no fas ús de les etapes *MEM* ni *ES*. En canvi les segones propaguen un valor, l'adreça de retorn, fins a *ES* el qual s'escriurà al registre número trenta-u.

Les instruccions de salt també es subdivideixen, independentment de la divisió anterior, en condicionals i incondicionals. Les primeres fan ús de la *ALU* per a

determinar si cal canviar el seqüenciament implícit, mentre que les segones no fan ús de la *ALU,* ja que salten sempre.

El bloc *EV* a l'etapa *ALU* és l'encarregat de decidir si cal saltar o no (depenent del tipus de salt i el resultat de la *ALU*) i també de triar l'adreça de destí, que depèn del tipus de salt.

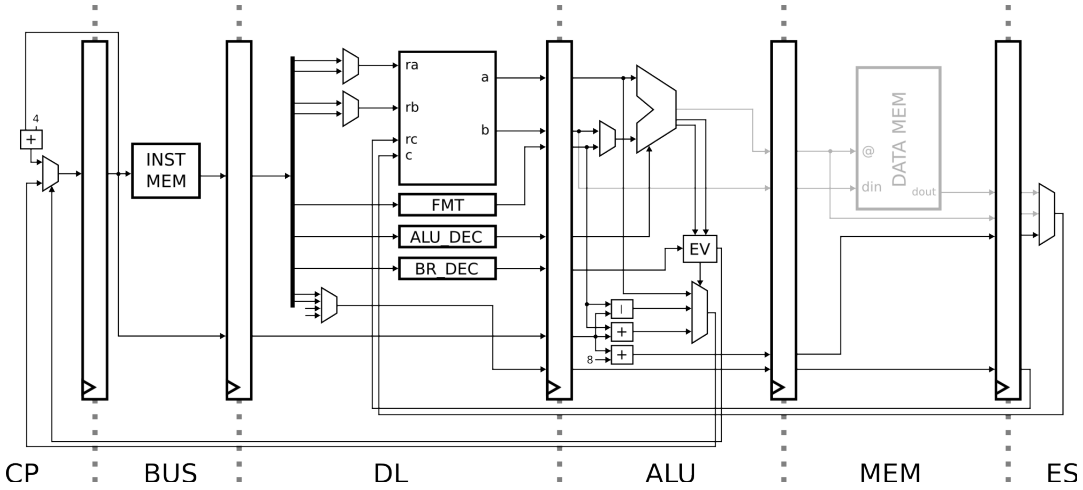


Figura 3.9: Ús del camí de dades per part de les instruccions de salt

4. Riscos d'interpretació segmentada

En la interpretació d'instruccions de forma segmentada apareixen riscos. Aquests riscos poden ser dividits en dos tipus segons la seva natura: riscos estructurals i riscos deguts a dades.

4.1 Riscos estructurals

Els riscos estructurals són aquells produïts per la utilització més d'una vegada d'un mateix recurs en un mateix cicle. En un processador lineal això succeeix quan dues etapes fan ús d'un mateix recurs. Per a evitar aquests riscos es pot afegir més recursos al processador de manera que en cap moment es faci ús del mateix recurs per part de dues etapes diferents.

En la microarquitectura que s'ha implementat no es produeixen riscos estructurals, ja que cada etapa té els seus recursos que no comparteix amb cap altra. En el cas particular de la memòria no hi ha tampoc cap risc ja que es tracta d'una arquitectura *Hardvard*, on les memòries de dades i d'instruccions són dos elements diferents.

4.2 Riscos de dades

Els riscos de dades apareixen a causa dels anomenats bucles hardware. S'entén per bucle hardware l'escriptura de posicions d'emmagatzemament en una etapa

posterior a la lectura d'aquestes [10]. En un processador *MIPS* només hi ha dos tipus de posicions d'emmagatzemament: registres i memòria de dades. Per tant només hi pot haver riscos de dades deguts a memòria i riscos de dades deguts a registres. Dels riscos deguts a registres en distingirem dos tipus segons els registres afectats: registres del banc de registres i el registre del comptador de programa.

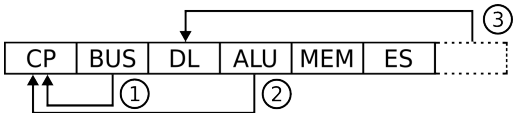


Figura 3.10: Bucles hardware i virtuals

A la *figura 3.10* es mostra un esquema amb els bucles hardware de la microarquitectura implementada. El bucle número 1 correspon a l'actualització del comptador de programa per part del seqüenciament implícit. El registre es llegeix de l'etapa *BUS*, s'incrementa el seu valor i s'escriu a l'etapa *CP*. El segon bucle correspon a l'actualització del comptador de programa per part d'una instrucció de salt (seqüenciament explícit). El valor es produeix a l'etapa *ALU* i s'escriu a l'etapa *CP*. El bucle número 3 correspon a l'escriptura d'un registre del banc de registres. Aquesta escriptura es produeix a l'etapa *ES* i la lectura a l'etapa *DL*. Tot i això, donat que l'escriptura no és efectiva fins al cicle següent de l'etapa *ES*, es representa a una etapa fictícia posterior a *ES* com es veu a la *figura 3.10*.

4.2.1 Riscos de dades deguts a memòria

En la microarquitectura implementada no es poden produir riscos de dades deguts a memòria, ja que els accessos a memòria es produeixen sempre en el mateix cicle i, per tant, no existeix cap bucle hardware. A la *figura 3.11* es representen dos accessos de memòria on es veu que no hi ha risc. La dada s'escriu al cicle 5 i es llegeix al cicle 6.

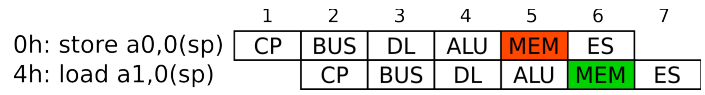


Figura 3.11: Exemple d'accés a memòria lliure de riscos de dades

4.2.2 Riscos de dades deguts al banc de registres

Ens centrarem en el tercer bucle de la *figura 3.10*: l'escriptura al banc de registres. Els registres s'escriuen sempre a l'etapa *ES* i es llegeixen a l'etapa *DL*, el que comporta una latència productor-ús igual a 4 cicles (es necessita tot el cicle per a escriure al registre a *ES*).

Aquesta latència entre lectura i escriptura provoca un risc de tipus *LDE* (Lectura Després d'Escriptura). El risc es produeix quan una instrucció vol llegir el valor d'un registre que encara no ha estat actualitzat per una instrucció anterior. A la *figura*

3.12 es mostra un exemple on la primera instrucció escriu un valor al registre *a0* i la instrucció següent llegeix aquest registre. La segona instrucció llegirà a un valor incorrecte, ja que encara no ha estat actualitzat per la instrucció anterior.

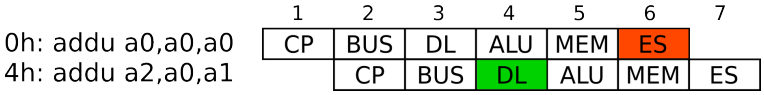


Figura 3.12: Exemplificació dels riscos de dades entre registres

4.2.3 Detecció dels riscos de dades deguts al banc de registres i actuació

Per a eliminar aquests riscos el que es fa és detectar quan una instrucció vol llegir el valor d'un registre que no està disponible encara i bloquejar la seva interpretació fins que el valor estigui disponible.

La detecció es fa comparant els identificadors de registres font de la instrucció que llegeix amb els identificadors de registre destí de les instruccions que es troben en procés d'interpretació i que escriuen als registres. El risc es detecta a l'etapa *DL* quan es disposa dels identificadors dels registres font.

L'actuació consisteix en bloquejar les instruccions a les etapes *CP*, *BUS* i *DL* impedint que progressin per la segmentació. Les instruccions a les etapes *ALU*, *MEM* i *ES* sí que progressen per la segmentació i eventualment escriuen al banc de registres fent desaparèixer el risc. A cada cicle de bloqueig s'introdueix una instrucció *nop* a l'etapa *ALU*, ja que com s'ha dit la instrucció a *DL* no progressa per la segmentació. La instrucció *nop* és una instrucció que no fa res, és a dir, no modifica ni la memòria ni els registres.

A la *figura 3.13* es mostra una interpretació correcta del programa anterior (*figura 3.12*). La segona instrucció queda retinguda a l'etapa *DL* a l'espera de que finalitzi la primera per a poder llegir el valor del registre *a0*. Es pot veure que s'introdueixen tres instruccions *nop* durant els cicles on es detecta el risc. Els cicles en els quals s'introdueix una *nop* a la segmentació direm que són cicles perduts, ja que no inicien cap instrucció nova. En l'exemple els cicles 4, 5 i 6 serien cicles perduts. En el cicle 7 el risc desapareix i es reempren la interpretació de la instrucció així com l'inici de noves instruccions.

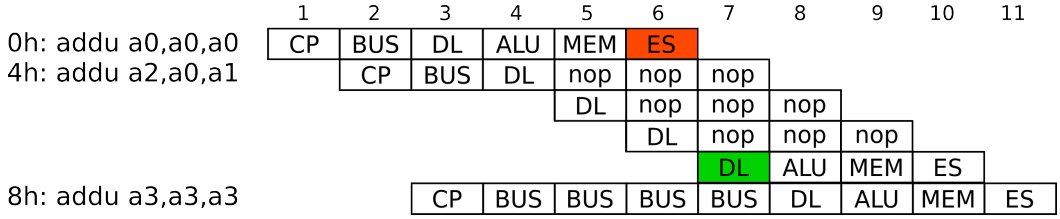


Figura 3.13: Eliminació de riscos de dades mitjançant la injecció de cicles d'espera

4.2.3 Curtcircuits al camí de dades

Amb l'objectiu d'evitar una pèrdua de cicles quan es produeix un risc *LDE* el que es fa es afegir els anomenats curtcircuits. Un curtcircuit és un camí afegit al camí de dades que permet l'accés a dades produïdes al llarg del camí de dades abans que aquestes s'escriguin al banc de registres. L'explicació és que, a la majoria d'instruccions, el valor que s'escriurà al banc de registres està disponible uns cicles abans d'arribar a l'etapa *ES*.

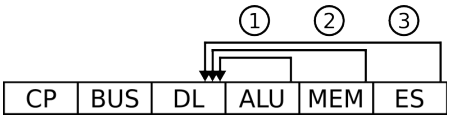


Figura 3.14: Curtcircuits del processador

Els curtcircuits s'implementen amb cables que van des de punts de producció de dades fins a l'etapa *DL*, on es llegeixen les dades. Una unitat lògica de control de curtcircuits indica a l'etapa *DL* si ha d'utilitzar el valor llegit del banc de registres o bé ha d'utilitzar el valor provinent d'un curtcircuit. A la *figura 3.14* es mostren els conjunts de curtcircuits que s'han implementat al processador. El conjunt de curtcircuits número 1 permet l'accés a dades produïdes per la *ALU*. El conjunt de curtcircuits número 2 permet l'accés a la dada llegida per la memòria o bé a la dada produïda per la *ALU* en el cicle anterior. El conjunt de curtcircuits número 3 permet accedir a la dada que s'està escrivint al banc de registres.

A la *figura 3.15* es mostra un programa que fa ús de 3 curtcircuits per a accedir a una dada que encara no s'ha escrit al banc de registres (registre *a0*) i, d'aquesta manera, reduir el nombre de cicles perduts. Al cicle 4 es fa ús d'un curtcircuit del primer conjunt, al cicle 5 s'utilitza un del segon conjunt de curtcircuits i al cicle 6 es fa ús d'un del tercer conjunt.

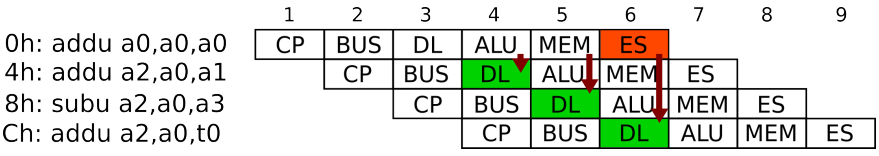


Figura 3.15: Exemple d'utilització dels curtcircuits del camí de dades

A la *figura 3.16* es mostra amb detall el canvi al camí de dades en les etapes *DL*, *ALU*, *MEM* i *ES* un cop s'han afegit els curtcircuits. Aquests estan formats per busos amb origen als punts de producció de dades fins a l'etapa *DL*. Es fa ús de multiplexors per a triar entre les diferents dades procedents d'etapes posteriors i la dada provinent del banc de registres. Els curtcircuits implementats permeten reduir el nombre de cicles perduts en la majoria de les situacions, però hi ha situacions en les que no és possible ja que encara no s'ha produït la dada necessària. En aquests casos es bloqueja la interpretació de la instrucció en *DL* fins que desapareix el risc.

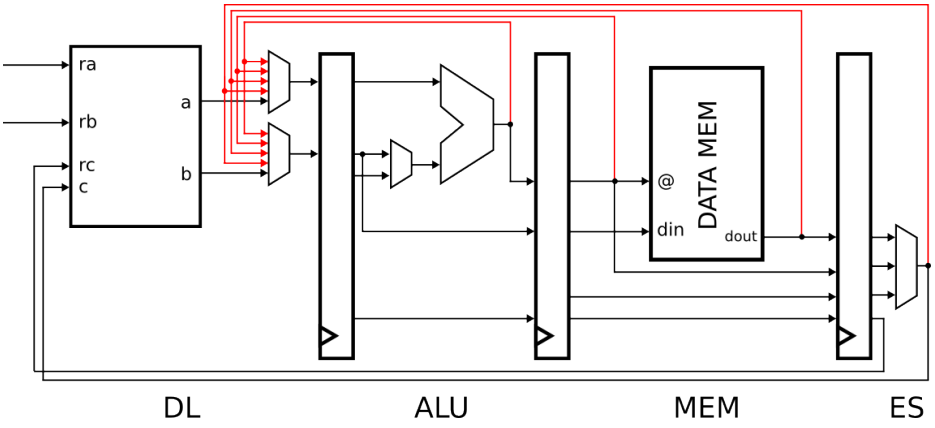


Figura 3.16: Esquema del camí de dades amb curtcircuits

4.3 Riscos de seqüenciament

Un risc de seqüenciament és un risc de dades degut al registre comptador de programa. El canvi del comptador de programa per part del seqüenciament implícit (bucle 1 de la *figura 3.10*) no provoca cap risc ja que el bucle té longitud 1 i el processador inicia una instrucció per cicle com a màxim.

Es produeix un risc de seqüenciament quan una instrucció canvia el comptador de programa de forma explícita degut a l'existència d'un bucle hardware (segon bucle hardware de la *figura 3.10*). Les instruccions que permeten canviar explícitament el comptador de programa són les instruccions de salt. En l'arquitectura *MIPS* aquestes instruccions estableixen el valor del comptador de programa de forma retardada, el que s'anomena “salt retardat” o *delayed branch*.

El salt retardat consisteix en fer efectiu el canvi del comptador de programa una instrucció després de la interpretació de la instrucció de salt. Això significa que la instrucció situada immediatament després del salt a la memòria d'instruccions s'interpreta sempre.

Al la *figura 3.17* es pot veure un llistat d'exemple. La instrucció situada immediatament després de la instrucció de salt (08h) s'anomena *delay slot* i acostuma a ser una instrucció que no generi cap salt (típicament aritmeticològica o d'accés a memòria).

```
000h: addiu s0,s1,3
004h: j 100h            # Instrucció de salt
008h: subu t0,t1,t2     # Delay slot
...
100h: li t1, 12         # Jump target
```

Figura 3.17: Exemple d'interpretació d'una instrucció de salt

En el cas dels salts condicionals cal tenir en compte que quan el resultat d'avaluar la condició sigui fals no es produirà un canvi en el comptador de programa. En

aquests casos cal continuar interpretant instruccions sense canviar el comptador de programa.

Per a interpretar totes les instruccions de salt i amb l'objectiu de reduir cicles perduts per aquests riscos s'ha implementat predicció de salt. La predicció de salt consisteix en suposar que la instrucció de salt no canviarà el comptador de programa i, en el cas que aquesta suposició no es compleixi, descartar les instruccions posteriors al *delay slot* que es trobessin en procés d'interpretació.

El bucle hardware d'actualització del comptador de programa es troba de l'etapa *ALU* fins a *CP*. Dit d'una altra manera el comptador de programa no s'escriu fins que la instrucció de salt ha arribat a l'etapa *ALU* ja que, entre altres motius, és en aquesta etapa on s'avalua la condició del salt pels salts condicionals. Quan això succeeix hi ha una instrucció a l'etapa *DL* i una altra a l'etapa *BUS*. En el cas de la instrucció en *DL* sabem que es tracta del *delay slot*, ja que és la instrucció immediatament posterior al salt. Però la instrucció que es troba a l'etapa *BUS* és una instrucció que no es desitja interpretar i, per aquest motiu, és necessari actuar per evitar-ne la seva interpretació.

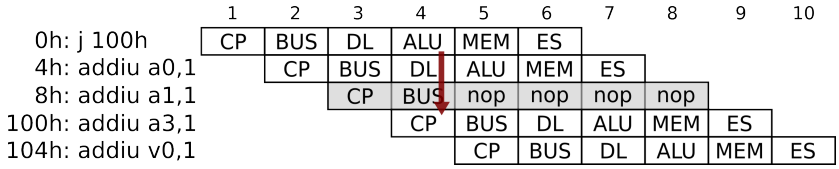


Figura 3.18: Exemple d'interpretació d'una instrucció de salt

A la *figura 3.18* es veu un exemple de salt. La primera instrucció salta incondicionalment a l'adreça 100h. La segona instrucció es tracta del *delay slot* de la primera i per tant s'executa, mentre que la tercera instrucció es descarta (es converteix en una *nop*). En el cicle 4 s'estableix el nou comptador de programa i la instrucció destí del salt entra a la segmentació.

5. Excepcions i interrupcions

En els processadors *MIPS* hi ha dos modes d'execució: usuari i privilegiat. La diferència entre els dos modes de funcionament és que el mode usuari només pot executar un subconjunt d'operacions del mode privilegiat. L'objectiu d'aquesta estratègia és protegir algunes de les operacions que pot realitzar un ordinador en un mode privilegiat i fer funcionar el processador en el mode no privilegiat (usuari) sempre que es pugui.

En un processador *MIPS* el mode privilegiat és l'únic que pot accedir als dispositius i a certes àrees de la memòria. D'aquesta manera s'evita que un programa d'usuari

mal programat pugui interferir amb els dispositius i provocar errors que afectin a la resta de programes. Aquesta divisió entre els dos modes facilita en gran part l'existència dels sistemes operatius.

5.1 Excepcions i interrupcions del processador

Una excepció és un esdeveniment que es produeix durant la interpretació d'instruccions en un processador i que té com a objectiu informar de problemes o situacions anòmales. Parlarem d'excepcions quan ens referim a esdeveniments síncrons amb el procés d'interpretació d'instruccions, és a dir, esdeveniments associats a la interpretació d'instruccions [11].

Es parla d'interrupció quan es produeix un esdeveniment en el que un perifèric demana l'atenció del processador de forma asíncrona, és a dir, d'una manera que no té relació amb el procés d'interpretació d'instruccions. Tot i això i degut a la similitud entre excepcions i interrupcions ens referirem als dos conjunts com a excepcions i només els diferenciarem en els casos on sigui necessari.

Les excepcions es distingeixen per la seva causa i a cada una se li associa un nombre [11]. A la *figura 3.19* es mostren les excepcions implementades al processador. Es mostra també l'etapa del processador segmentat on es realitza la detecció de l'excepció.

Excepció	Codi	Etapa	Descripció
Interrupció	0	N/A (*)	Es produeix una interrupció externa
Address error (read)	4	BUS/ALU	Accés de lectura a memòria no alineat
Address error (write)	5	ALU	Accés d'escriptura a memòria no alineat
Bus error (inst fetch)	6	BUS	Accés a memòria d'instruccions no vàlid
Bus error (load/store)	7	ALU	Accés a memòria de dades no vàlid
Syscall	8	DL	S'ha executat la instrucció *syscall*
Break	9	DL	S'ha executat la instrucció *break*
Instrucció reservada	10	DL	La instrucció no té una codificació vàlida
Overflow	12	ALU	Es produeix desbordament en un add/sub

Figura 3.19: Taula d'excepcions implementades al processador

(*) Les interrupcions no estan associades a cap etapa, ja que no són provocades per una instrucció sinó per elements externs a la interpretació d'instruccions

5.2 Coprocessador 0

Per a implementar tots els mecanismes necessaris per a la gestió d'excepcions calen una sèrie de registres on guardar dades relacionades amb aquests mecanismes. Aquests registres es troben en un banc de registres en l'anomenat coprocessador 0. A la *figura 3.20* es mostra els registres que es fan servir en el processador [11]. Se'n parlarà més detalladament a continuació.

Registre	Num	Descripció
BadVAddr	8	Adreça de l'accés incorrecte a memòria
Counter	9	Comptador de cicles de rellotge
Compare	11	Registre de compració del timer
Status	12	Estat del processador
Cause	13	Causa de l'última excepció/interrupció
EPC	14	Adreça on s'ha produït l'última excepció

Figura 3.20: Registres implementats al coprocessador 0

Per a la lectura i escriptura dels registres es fa ús de les instruccions llistades a la *figura 3.21*. Aquestes instruccions només poden ser executades en mode privilegiat.

Instrucció	Descripció
`mtc0 reg,regX`	Mou el valor d'un registre al registre del CP0
`mfc0 reg,regX`	Mou el valor d'un registre del CP0 a un registre general

Figura 3.21: Instruccions implementades per al coprocessador 0

5.3 Gestió d'excepcions

Quan es detecta una excepció es procedeix a gestionar-la seguint els següents passos [11]:

1. La unitat de control del processador escriu al registre *EPC* el valor del comptador de programa de la instrucció que ha provocat l'excepció. En cas que l'anterior instrucció fos un salt s'escriu l'adreça de la instrucció anterior.
2. El control del processador escriu al registre *Cause* el codi de l'excepció que s'ha produït.
3. El control del processador modifica el registre *Status* per inhibir les interrupcions i entrar en mode privilegiat.
4. El control del processador garanteix que totes les instruccions anteriors a la que ha produït l'excepció s'han interpretat i que les següents (inclosa la causant) no han produït cap efecte, com si no s'haguessin interpretat.
5. El control del processador estableix el nou comptador de programa a una adreça fixa, on es troba la primera instrucció de la rutina de servei a les excepcions.

A la *figura 3.22* es mostra un exemple on la instrucció *add* provoca una excepció. Tot i que la detecció es produeix al cicle 4 (etapa *ALU*) no és fins el cicle 5 (etapa *MEM*) on s'actua. Es descarten totes les instruccions incloent la que ha provocat l'excepció i es canvia el *CP* a la primera instrucció de la rutina de servei d'excepcions. Tot i que no es mostra en la figura, en el cicle 5 s'escriu als registres del coprocessador 0 les dades sobre l'excepció.

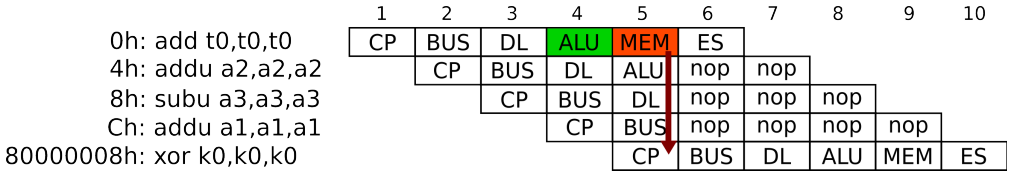


Figura 3.22: Codi que mostra la interpretació d'instruccions durant la gestió d'una excepció.

5.4 Gestió per part del nucli

Segons el punt anterior un cop s'ha produït l'excepció el processador es troba en mode privilegiat i amb el comptador de programa apuntat a una adreça fixa. És a partir d'aquesta adreça on el programador de sistema (del nucli) haurà d'escriure codi que gestioni l'excepció i, a continuació, retorni al mode d'usuari.

El codi situat a partir d'aquesta adreça de memòria s'anomena gestor d'excepcions (*exception handler*) i s'encarrega de realitzar les tasques necessàries per al tractament de l'excepció. La rutina pot accedir a les dades de l'excepció (llegint els registres del coprocessador 0) per tal de conèixer la causa de l'excepció amb detall.

Finalment per a retornar al mode usuari fa ús del registre *EPC*, que conté l'adreça de retorn, així com la instrucció *rfe*. Aquesta instrucció rehabilita la detecció d'excepcions i torna a mode usuari.

5.5 Implementació de les excepcions

Les excepcions (excloent les interrupcions) es tracten en ordre de programa per a seguir el model d'interpretació sèrie. És a dir, donades dues instruccions que produeixen una excepció s'ha de tractar primer sempre l'excepció produïda per la instrucció més antiga. Donat que les excepcions es detecten en etapes diferents (segons la seva causa, veure *figura 3.19*) cal gestionar les excepcions en una etapa on es pugui garantir aquest ordre.

S'ha pres la decisió de tractar totes les excepcions en una mateixa etapa: la etapa *MEM*. Cada vegada que es produeix una excepció aquesta es propaga a l'etapa següent fins arribar a *MEM*, on es gestiona.

A la *figura 3.23* es pot veure un exemple on dues instruccions produeixen una excepció. El seu tractament a l'etapa *MEM* garanteix que es gestionin sempre en ordre de programa. De no ser així es gestionaria abans l'excepció de la segona instrucció que de la primera.

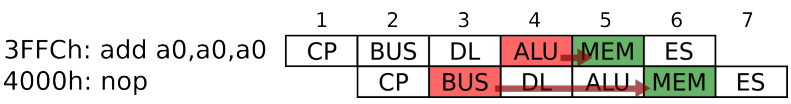


Figura 3.23: Exemple d'excepcions detectades en etapes diferents i gestionades a la mateixa etapa

L'entitat que implementa el coprocessador 0 s'ha situat a l'etapa MEM pels motius que s'ha esmentat anteriorment. A la *figura 3.24* es pot veure el camí de dades amb el coprocessador.

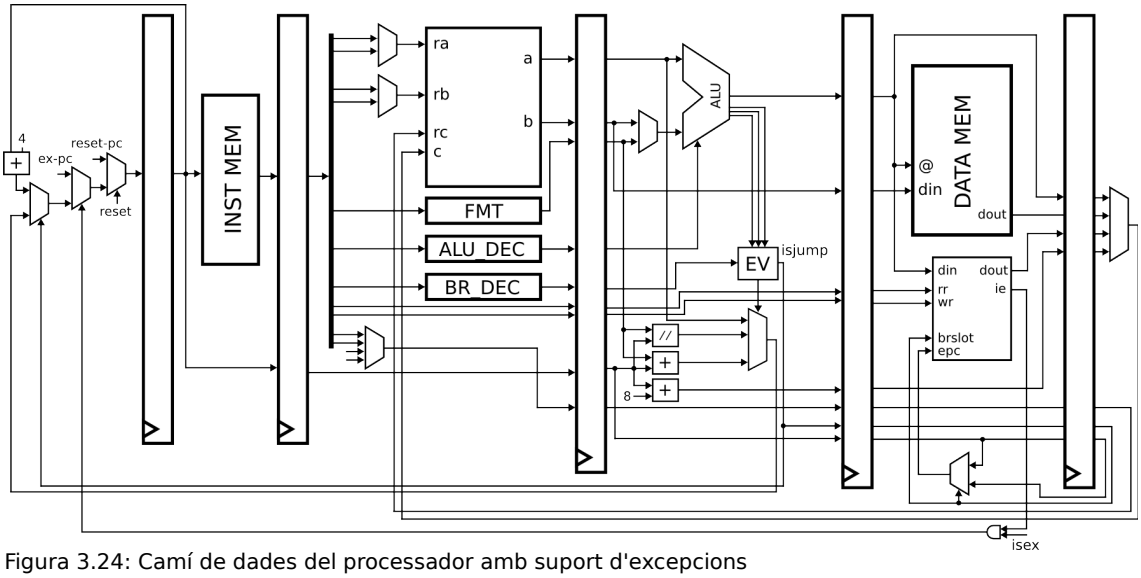


Figura 3.24: Camí de dades del processador amb suport d'excepcions

6. Memòria

Les adreces de memòria en un processadors *MIPS* tenen una mida de 32 bits. Això permet adreçar fins a 4GB de memòria fent ús d'instruccions *load*/*store*. Ara bé, no totes les adreces de memòria accessibles es corresponen amb un bloc de memòria físic. Distingirem entre adreces lògiques i físiques. Les adreces lògiques són les adreces que el programador especifica com arguments de les instruccions *load*/*store* i les que fa servir el processador en els seus càlculs. En canvi les adreces físiques són aquelles adreces que es fan servir per accedir a la memòria físicament (al component de memòria).

La separació entre memòria física i física permet molta llibertat a l'hora de dissenyar un computador. Podria ser que dues adreces lògiques accedissin realment a la mateixa posició de memòria física. O al revés, que una mateixa adreça lògica

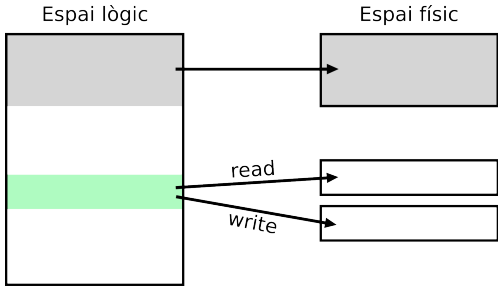


Figura 3.25: Exemple de correspondència entre l'espai lògic i físic de memòria

accedís a dues posicions de memòria física diferents (*figura 3.25*).

Els processadors *MIPS* aprofiten aquesta separació entre memòria física i virtual per a comunicar-se amb l'exterior, és a dir, amb els perifèrics. La implementació d'aquesta comunicació es fa reservant un conjunt d'adreces lògiques que corresponen a un controlador d'entrada sortida enlloc d'una memòria. Quan es fa una lectura o escriptura les adreces esmentades el controlador accedeix als dispositius per a llegir o escriure (*figura 3.26*).

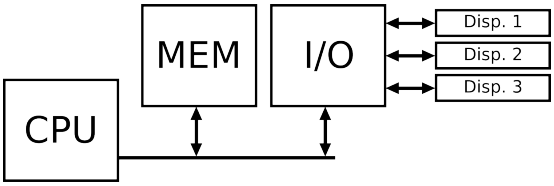


Figura 3.26: Esquema de l'accés a memòria per part del processador

Els dispositius poden ser tot tipus d'aparells amb una interfície digital: teclat, ratolí, monitor, etc. A la *figura 3.27* es poden veure les adreces associades als dispositius implementats al computador.

Adreça base	Mida (bytes)	Descripció
0xC0000000	4	Registre del display de LEDs
0xC0000004	4	Registre dels interruptors
0xC0000008	4	Registre del controlador de teclat
0xC0001000	3200	Memòria del controlador de pantalla

Figura 3.27: Espai de memòria dels controladors de dispositius

Per evitar un mal ús dels dispositius per part dels programes que executa el processador es protegeix el seu accés permetent-ne només l'accés quan el processador es troba en el mode privilegiat. En la microarquitectura implementada s'ha protegit la meitat de l'espai virtual de memòria. Les adreces que amb el bit de major pes igual a 1 són adreces que només es poden accedir en mode privilegiat. Un accés a l'espai protegit per part del mode usuari provoca una excepció.

Capítol 4

Implementació del processador

L'objectiu d'aquest capítol és descriure en detall quines característiques s'han implementat al processador i com s'ha realitzat aquesta implementació. Es detallaran els blocs més importants del processador i es farà espacial èmfasi en els perifèrics. Finalment per tancar el capítol es realitzarà un anàlisi del temps de cicle del retard de les diferents unitats funcionals del processador.

1. Característiques implementades

Els processadors MIPS formen una gran família d'arquitectures que ha evolucionat al llarg del temps. Existeixen diferents *ISA*s: *MIPS I*, *II*, *III*, *IV* i *MIPS32/64* les quals afegeixen noves instruccions i funcionalitats a les anteriors de manera que mantenen la compatibilitat binària [12]. A la *figura 4.1* es representen les 4 primeres *ISA* en un diagrama de *Venn*.

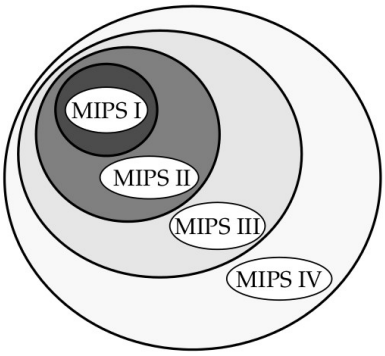


Figura 4.1: Diagrama de les *ISA MIPS*

1.1 Conjunt d'instruccions

S'ha implementat un subconjunt de la *ISA* del *MIPS I*. A l'*annex A* es pot veure un llistat detallat de les instruccions implementades. Aquestes es poden resumir en:

- Accessos a memòria a nivell de byte, *halfword* (2 bytes) i *word* (4 bytes).

- Operacions aritmeticològiques.
- Operacions de salt i crida.
- Operacions de crida a sistema i retorn d'excepció.

1.2 Arquitectura de memòria

El processador disposa de dos blocs de memòria disjunts. Per una banda té una memòria per a guardar les instruccions del programa i per altre banda té un bloc de memòria per a les dades del programa. Aquesta arquitectura rep el nom d'arquitectura *Harvard* [13].

L'objectiu de la divisió de memòria en memòria de dades i instruccions és eliminar els riscs estructurals deguts a aquesta. Les conseqüències que comporta aquesta divisió son principalment dues: les instruccions d'accés a memòria no poden llegir ni escriure la memòria d'instruccions i no es poden executar programes carregats a memòria de dades.

S'ha decidit per simplicitat no implementar traducció d'adreces, és a dir, els accessos a memòria es realitzen sense traducció d'adreces lògiques a físiques. Tot i això sí s'ha implementat protecció d'una part de l'espai lògic d'adreces com a memòria només és accessible pel nucli (mode privilegiat).

1.3 Procés d'interpretació

El procés d'interpretació d'instruccions està segmentat. Aquest disposa d'una unitat de control de riscos que lliura al programador de la gestió de riscos de dades. El processador *MIPS I* requeria que els programes gestionessin els riscos de dades incloent instruccions d'espera (*nop*s) després dels accessos a memòria ja el processador no eliminava els riscos. En aquest punt el processador es diferencia del *MIPS I* per lliurar al programador d'aquesta responsabilitat.

Per a millorar les prestacions i evitar cicles d'espera el processador disposa de curtcircuits. A més les instruccions de salt implementen l'anomenat *delayed branch*, el que vol dir que executen sempre la instrucció immediatament posterior al salt. Això contribueix també a reduir el nombre de cicles perduts per riscs de seqüenciament. La majoria d'instruccions del processador s'executen sense perdre cicles. Es pot consultar el nombre de cicles perduts per cada instrucció a l'*annex A*.

2. Síntesi en *FPGA*

La implementació del processador en *VHDL* requereix certes consideracions per a poder ser sintetitzada a una *FPGA*. El procés d'adaptació del programa *VHDL* a la síntesi depèn del dispositiu *FPGA* triat però sobretot depèn de l'entorn de síntesi escollit. En aquest cas donat que la *FPGA* és de la marca *ALTERA* caldrà fer ús de les eines que posa aquesta a disposició del públic: el programa *Quartus*.

El software *Quartus II* és un entorn de desenvolupament de circuits digitals compatible amb el llenguatge *VHDL*. Disposa d'un compilador de *VHDL* així com d'un enrutador i un sintetitzador per les *FPGA*s compatibles. Permet, a més, l'ús d'esquemàtics per al disseny de circuits mitjançant un editor que incorpora.

2.1 Particularitats de la implementació

La implementació fent ús de *Quartus* imposa al disseny certs requeriments no funcionals que s'exposen a continuació.

2.1.1 Síntesi de memòries

Les unitats de memòria RAM s'implementen fent ús de les eines de que disposa el programa *Quartus* enlloc de fer ús d'entitats *VHDL*. El software disposa d'un programa capaç de crear memòries *RAM* i *ROM* i generar un fitxer *VHDL* que es pot fer servir per a crear una instància al disseny quan sigui necessari [14]. La raó d'aquest procediment és que aquesta *FPGA* disposa de blocs específics per a memòria. Aquests blocs s'anomenen *M9K* i permeten la síntesi de memòria més eficient, ja que de no existir caldria implementar la memòria amb *LUT*s. A la *figura 4.2* es mostra una imatge del programa emprat per a generar entitats de memòria.

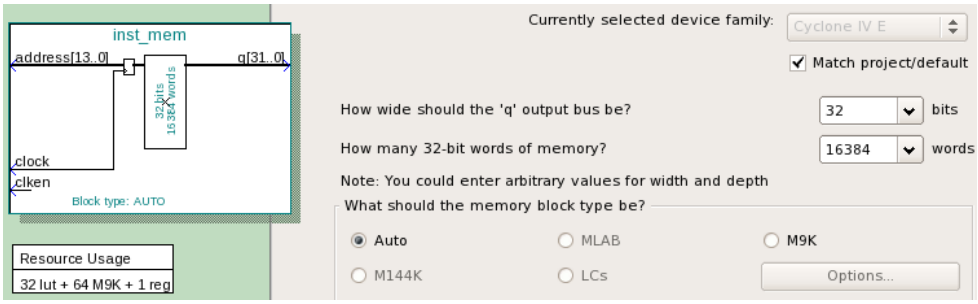


Figura 4.2: Interfície del programa Quartus per a crear entitats de memòria

2.1.2 Connexions d'entrada-sortida

L'entrada-sortida del processador representa connectar senyals d'entitats a *pins* (contactes físics) de la *FPGA*. Això s'especifica creant una entitat i especificant que les seves entrades i sortides siguin *pins* de la *FPGA*. Aquesta entitat que connecta

amb l'exterior s'anomena entitat superior (*top-level entity*) i només en pot existir una en cada disseny. Així doncs tota la resta d'entitats estan contingudes en aquesta.

Les senyals que es connectaran a l'exterior són les associades als perifèrics (pantalla, teclat, *display* i interruptors), el botó de *reset* i la senyal de rellotge. Es pot trobar un llistat detallat d'aquestes senyals a l'*annex C*.

3. Funcionament del processador

Els processadors es poden dividir en dues grans unitats: el camí de dades i la unitat de control. La primera s'encarrega de realitzar els càlculs i tasques que requereixen les instruccions interpretades, mentre que la segona és l'encarregada de governar el camí de dades.

En el processador implementat la unitat de control es troba distribuïda en diferents components cosa que permet que el processador sigui més eficient en la interpretació d'instruccions respecte a una lògica de control centralitzada. El principal motiu és que la lògica de control centralitzada requereix accés a senyals molt allunyades físicament.

El processador està segmentat es sis etapes. Cada una d'aquestes etapes fa ús d'unitats funcionals diferents per tal de realitzar les seves funcions. A la *figura 4.3* es pot veure un diagrama amb les etapes i les unitats que formen cada una, així com les unitats de control de curtcircuits i bloqueig, que pertanyen a la unitat de control.

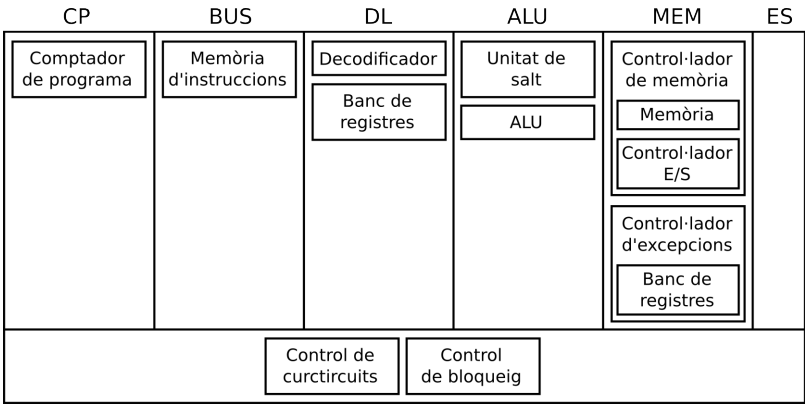


Figura 4.3: Esquema d'etapes amb les seves unitats funcionals

A continuació es descriuen les unitats de cada etapa. A l'*annex B* es pot trobar la descripció detallada i el codi *VHDL* associat a cada unitat.

3.1 Comptador de programa

Aquesta unitat és la responsable de determinar la propera instrucció a interpretar. L'adreça que calcula es propagarà fins la memòria d'instruccions on es cercarà la instrucció indicada.

La unitat rep com entrades les sortides del controlador d'excepcions i de la unitat de salt. Per defecte el comptador de programa assumeix seqüenciament implícit, pel que simplement incrementa el comptador de programa a cada cicle per avançar l'adreça a la propera instrucció. Si la unitat de salt indica un canvi en el seqüenciament aleshores estableix els comptador de programa al valor que la unitat de salt especifica. Això es realitza quan la unitat de salt interpreta una instrucció de salt. En cas que el controlador d'excepcions indiqui una excepció el comptador de programa passa a valer 0x80000008.

3.2 Memòria d'instruccions

Aquesta unitat està formada per dos blocs de memòria *ROM* (memòria de només lectura) i una lògica de control. Els dos blocs de memòria emmagatzemen les instruccions del programa d'usuari i del nucli, que es troben a adreces diferents.

Les memòries *ROM* han estat especificades fent ús de les eines proporcionades pel programa *Quartus*. Aquest permet definir memòries *ROM* i carregar-los dades. Les memòries que sintetitza el programa tenen una particularitat i és que el port d'adreça d'entrada està registrat. Això significa que l'adreça que s'especifica a l'entrada de la memòria no es fa efectiva fins que la memòria rep un flanc ascendent a la seva senyal de rellotge. Per aquest motiu entre les etapes *CP* i *BUS* hi ha certes senyals (les adreces de memòria) que no es transmeten a través de registres sinó que es connecten amb cables directament.

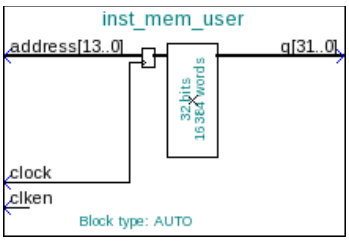


Figura 4.4: Símbol d'una memòria ROM on s'aprecia el port d'adreça registrat

3.3 Descodificador

El descodificador és una de les unitats més importants del processador. La seva funció és generar un conjunt de senyals de control que governaran les altres unitats a partir de la instrucció que rep de la memòria. Aquestes senyals es propaguen per

la segmentació fins arribar a l'etapa on són necessàries. Les podem dividir en els conjunts següents:

- Senyals d'accés al banc de registres: Els nombres dels dos registres font i del registre de destí.
- Senyals de control de la *ALU*: Indiquen quina operació es realitza a l'*ALU*.
- Senyals de descodificació de literals: Els literals immediats de 16 o 5 bits amb extensió de signe.
- Senyals d'interpretació de salts: El tipus de salt i els bits de salt relatius al comptador de programa.
- Senyals d'accés a memòria: Bits de permís d'escriptura i mida de l'accés a memòria.
- Senyals de suport a les excepcions: Proporciona bits que indiquen si la instrucció és vàlida i si pot provocar algun tipus d'excepció. També proporciona bits que indiquen si la instrucció fa accessos al banc de registres del coprocessador o bé si cal retornar d'una excepció.
- Senyals d'encaminament de dades: Bits que indiquen al camí de dades quins camins de dades interconnectar. Especifiquen quina unitat produeix cada dada.

Tot i la gran quantitat de dades que genera el descodificador la seva implementació és relativament senzilla, ja que les instruccions del *MIPS* estan codificades amb l'objectiu de permetre una ràpida descodificació.

3.4 Banc de registres

El banc de registres conté 31 registres de 32 bits cada un que es fan servir per a emmagatzemar els resultats de les operacions que realitza la unitat de procés. Disposa de dos ports de lectura que permeten l'accés al valor de dos registres de forma simultània. També disposa d'un port d'escriptura que permet escriure el contingut d'un registre.

Els registres s'identifiquen amb un nombre de 5 bits, pel que es té 32 possibles registres. El registre número zero és diferent als altres registres ja que sempre té el valor de zero. Qualsevol accés de lectura d'aquest registre dóna el valor zero i qualsevol accés d'escriptura no té efecte.

3.5 Unitat aritmeticològica (ALU)

L'*ALU* és la unitat encarregada de realitzar els càlculs aritmètics i lògics requerits per algunes instruccions. Consta de dues entrades de 32 bits corresponents als dos operands i una sortida de 32 bits corresponent al resultat del càlcul. Disposa d'una entrada per a seleccionar l'operació que s'ha de realitzar, que està controlada pel descodificador (*figura 4.5*).

A banda d'aquestes senyals disposa d'una sortida de quatre bits que indiquen propietats sobre les senyals d'entrada. Aquestes senyals es propaguen a la unitat de salt per a avaluar els salts condicionals. A l'*annex B* es hi ha una descripció detallada de les operacions que realitza i de les senyals auxiliars de sortida.

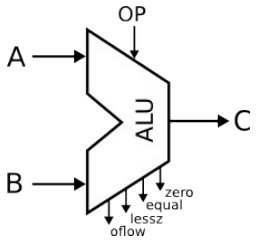


Figura 4.5: Esquema de la ALU

3.6 Unitat de salt

La unitat de salt és la unitat que interpreta les instruccions de salt i decideix si cal canviar el seqüenciament implícit i quina és l'adreça a la que cal saltar.

Per una banda conté un bloc que avalua si la instrucció canvia el seqüenciament implícit. Això es compleix si la instrucció és un salt incondicional o bé si és condicional i compleix les condicions especificades (informació indicada pels bits de sortida de la *ALU* esmentats al punt anterior).

Paral·lelament es calcula l'adreça on es realitza el salt depenent del tipus de salt. Hi ha tres tipus de salts:

- Salts especificats per registres: S'emmagatzema al comptador de programa el valor del registre especificat per la instrucció.
- Salts relatius al comptador de programa: Es calcula l'adreça a emmagatzemar a partir de l'adreça de la instrucció i d'un literal codificat a la instrucció.
- Salts absoluts: S'emmagatzema al comptador de programa el valor especificat per un literal codificat a la instrucció.

Finalment la unitat calcula, també de forma paral·lela, l'adreça de retorn del salt. Si el salt és una crida es guardarà al registre número trenta-u l'adreça de la instrucció

a la que cal retornar un cop finalitzada la crida. Aquesta instrucció és troba situada a l'adreça de la instrucció de salt més 8 bytes. Aquests 8 bytes corresponen als 4 bytes de la instrucció de salt més els 4 bytes de la instrucció situada al *delay slot*.

3.7 Controlador de memòria

El controlador de memòria és una unitat que conté les diverses subunitats del sistema de memòria. Les seves entrades i sortides són les típiques d'una memòria: una entrada d'adreces, busos de dades d'entrada i sortida i senyals d'entrada de control: permís d'escriptura i mida de l'accés a memòria (que pot ser 1, 2 o 4 bytes). Addicionalment té una entrada que indica si cal realitzar extensió de signe en els accessos a byte o *halfword* (dos bytes), ja que les instruccions de lectura disposen d'un bit que permet estendre el signe de la dada llegida.

Les unitats que conté són un bloc de memòria *RAM* i un controlador d'entrada-sortida. La unitat de memòria conté les dades d'usuari i de sistema associades als seus corresponents espais d'adreces lògiques. El controlador d'entrada-sortida és un bloc que conté tota la lògica per a l'accés als dispositius del computador. A la *figura 4.6* es pot veure el contingut del controlador a nivell de blocs.

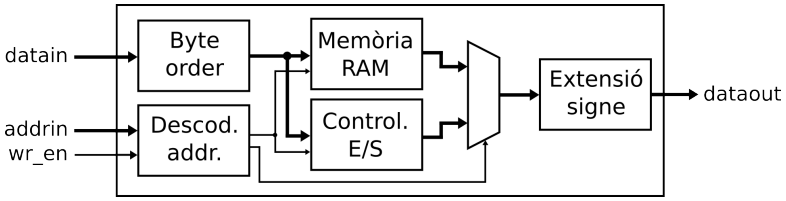


Figura 4.6: Diagrama de blocs del controlador de memòria

La dada d'entrada passa a través d'un bloc encarregat de desplaçar els bits adequadament pel cas d'accés a nivell de byte o *halfword*. El descodificador d'adreça indica quin bloc realitzarà l'accés transferint-li el permís d'escriptura i prohibint l'escriptura a la resta de blocs. Finalment es tria la sortida de la memòria fent ús del descodificador d'adreces i d'un multiplexor i es tracta l'extensió de signe en cas que sigui necessari.

3.7.1 Memòria RAM

Les unitats de memòria *RAM* s'implementen fent ús de les eines de que disposa el programa *Quartus*. Aquestes memòries tenen la particularitat d'estar registrades, és a dir, totes les seves entrades passen a través d'un registre governat per una senyal de rellotge. Tal com es pot veure a la *figura 4.7* l'adreça, la dada i el permisos d'escriptura passen a través d'un registre.

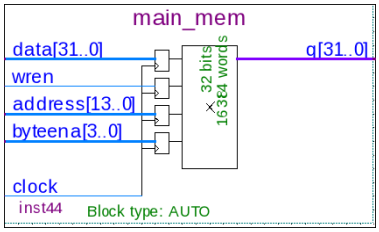


Figura 4.7: Símbol d'una memòria RAM

3.7.2 Controlador d'entrada-sortida

El controlador d'entrada-sortida disposa de les mateixes entrades i sortides que una memòria convencional. Aquest s'encarrega d'escriure o llegir dades dels dispositius del computador. Simplement conté subcontroladors per a cada un dels dispositius als que dóna servei. A la *figura 4.8* es poden veure els tres controladors de dispositius i la seva interfície amb el controlador.

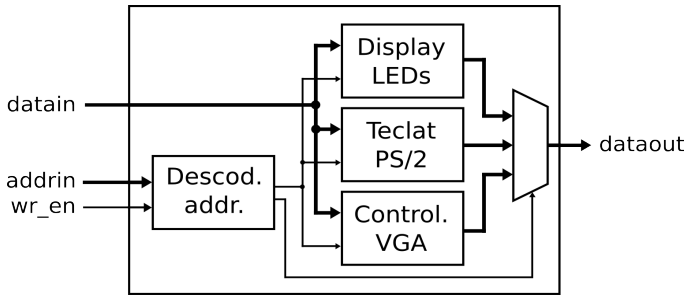


Figura 4.8: Diagrama de blocs del controlador d'entrada-sortida

3.8 Controlador d'excepcions

El controlador d'excepcions és la unitat responsable de l'actuació quan es produeix una excepció així com de la finalització d'una excepció. Conté el banc de registres del coprocessador 0, on s'emmagatzemen les dades relatives al tractament d'excepcions.

Disposa de les entrades convencionals d'un banc de registres: un port de lectura i un d'escriptura amb les seves corresponents entrades d'adreces i de dades. A més disposa de senyals d'entrada per notificar interrupcions i excepcions així com informació relativa a la instrucció que s'està executant. Com a sortides té el permís d'excepcions i el *flag* d'interrupció.

El controlador d'excepcions incorpora el rellotge de sistema, que és un dispositiu intern capaç de generar interrupcions periòdicament. Aquest dispositiu fa ús de registres del coprocessador 0 per al seu funcionament i, per aquest motiu, s'ha integrat al controlador d'interrupcions. Més endavant es parlarà amb més detall del tractament de les excepcions.

3.9 Controlador de curtcircuits

La unitat de control dels curtcircuits té la responsabilitat de controlar els multiplexors que implementen els curtcircuits (*figura 3.16*). Quan aquesta unitat detecta una dada no disponible al banc de registres calcula quin curtcircuit ha de fer servir per tal d'accedir al valor.

Per a la seva implementació es fan servir els identificadors de registres de lectura de l'etapa *DL* i l'identificador del registre d'escriptura de les etapes *ALU* i *MEM*. La unitat compara aquests valors per a determinar el curtcircuit a fer servir. Addicionalment fa ús de bits de validesa generats pel descodificador que indiquen a quina etapa es produeix el valor que s'escriu al banc de registres.

3.10 Controlador de bloqueig

El controlador de bloqueig controla les senyals de bloqueig i d'injecció de *nops* del camí de dades. S'encarrega de detectar els riscs de dades i seqüenciament i eliminar-los. Actua en tres possibles situacions:

- Risc de dades: Es bloquegen les etapes *CP*, *BUS* i *DL* i s'injecta una *nop* a l'etapa *ALU*. A mesura que *ALU*, *MEM* i *ES* progressen desapareix el risc.
- Risc de seqüenciament: S'injecta una *nop* a l'etapa *BUS* per tal de descartar la instrucció que s'havia predit.
- Risc de dades en una instrucció situada a un *delay slot*: Quan es produeix un risc de dades en una instrucció situada immediatament després d'un salt cal retardar el salt. La instrucció de salt es troba en etapa *ALU* (etapa on s'estableix el valor del comptador de programa) de manera que si es bloqueja *CP* com a conseqüència del risc però no es bloquegés *ALU* no es faria efectiu el salt. Es bloquegen les etapes *CP*, *BUS*, *DL* i *ALU* i s'injecta una *nop* a l'etapa *MEM* fins que desapareix el risc.

4. Excepcions i interrupcions

Distingirem a partir d'aquest punt entre excepcions i interrupcions, entenent les primeres com excepcions síncrones i les segones com excepcions asíncrones. En el processador s'ha implementat un conjunt d'excepcions d'entre totes les disponibles i dues fonts d'interrupció.

4.1 Excepcions

Les excepcions implementades es descriuen a continuació. Un processador *MIPS*

disposa de més excepcions, però en el nostre cas no tenen sentit i s'ha desestimat la seva implementació.

- Crida a sistema: Excepció que es produeix quan la instrucció que s'està interpretant és una instrucció de crida (*syscall*).
- *Breakpoint*: Excepció produïda per la interpretació d'una instrucció de *breakpoint* (*break*).
- Instrucció no vàlida: La instrucció que s'ha interpretat no té una codificació vàlida (segons la *ISA* del *MIPS I*) o bé no es permet la seva interpretació (interpretació d'instruccions privilegiades en mode usuari, per exemple).
- Desbordament (*overflow*): La instrucció interpretada ha executat una operació aritmètica que no és representable en 32 bits. Aquesta excepció només la produeixen certes instruccions tals com les sumes i restes.
- Error de bus d'instruccions: Es produeix aquesta excepció quan s'intenta accedir a memòria d'instruccions per a llegir una instrucció i el comptador de programa es troba apuntant a una adreça no permesa.
- Error de bus de dades: Es produeix aquesta excepció quan s'intenta accedir a memòria de dades per a llegir o escriure una dada i l'adreça que realitza l'accés no és vàlida.
- Error d'adreça (lectura): Error produït quan s'intenta llegir una dada de la memòria de dades o d'instruccions i l'adreça no està correctament alineada.
- Error d'adreça (escriptura): Error produït quan s'intenta escriure una dada a la memòria de dades i l'adreça no està correctament alineada.

Cada tipus d'excepció es detecta a una certa etapa i es propaga fins l'etapa *MEM* on serà gestionada. Aquesta propagació es realitza fent ús d'un mecanisme que encadena les diferents fonts d'excepció propagant-ne sempre la primera que es produeix. D'aquesta manera si es produeix més d'una excepció en el procés d'interpretació d'una instrucció només es propaga la més antiga. A la *figura 4.9* s'il·lustra el procés, on cada bloc de la cadena només pot produir una excepció si no se n'ha produït una abans.

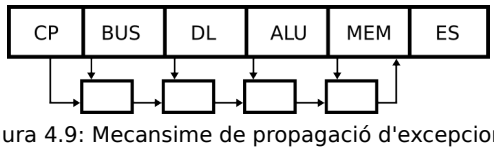


Figura 4.9: Mecansime de propagació d'excepcions

4.2 Interrupcions

En un processador *MIPS* es disposa de 8 possibles fonts d'interrupció. Cada una d'aquestes fonts d'interrupció té assignada un bit al registre d'estat del processador i un bit al registre d'excepció. En el registre d'estat s'especifica si la interrupció està permesa o no (màscara), mentre que al registre d'excepció s'indica si la interrupció s'ha produït o no (*flag*). Quan es produeix una interrupció el dispositiu estableix el bit de *flag* corresponent al valor de 1 i, si el corresponent bit d'habilitació també pren valor 1, es produeix una interrupció [11]. A les *figures 4.10* i *4.11* es mostra la descripció del contingut del registres esmentats.

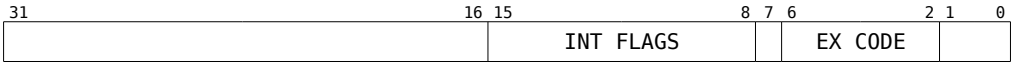


Figura 4.10: Descripció del registre d'excepció (*Cause*)

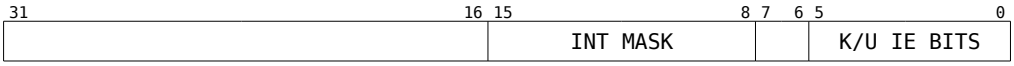


Figura 4.11: Descripció del registre d'estat (*Status*)

Donat que només hi ha dos possibles fonts d'interrupció només calen dos bits per a implementar-les. Es farà servir el 8è bit (de major pes) per a la interrupció de rellotge i el 7è bit per a les interrupcions externes. Per defecte (després d'un *reset*) els bits dels dos registres anteriors prenen el valor de zero, de manera que s'inhibeixen totes les interrupcions i es netegen tots els bits de *flag* d'aquestes.

El mecanisme de tractament d'interrupcions s'encarrega de produir una excepció (de tipus interrupció) quan detecta una interrupció. Per a fer-ho realitza una operació *and* bit a bit dels 8 bits de màscara amb els 8 bits de *flags*. Els bits resultants d'aquesta operació s'acumulen fent ús d'una *or* i la senyal resultant indica si hi ha alguna interrupció permesa pendent. A la *figura 4.12* es pot veure un diagrama que il·lustra aquesta operació.

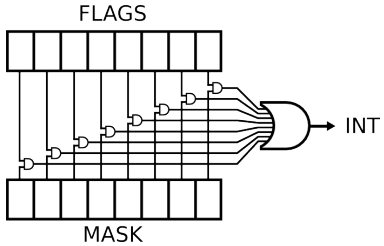


Figura 4.12: Generació de la senyal d'interrupció (INT)

4.3 Modes d'execució

Les excepcions permeten canviar el flux d'execució per a informar d'un comportament anòmal o bé d'algun esdeveniment associat a un dispositiu. A més d'això també canvien el mode l'execució del processador fent que aquest canvii a

mode privilegiat. Com s'explica al capítol anterior l'única manera de passar de mode usuari a mode privilegiat (o mode sistema) és fent ús d'una excepció [11].

El mode de treball del processador (sistema o usuari) està indicat per un bit del registre d'estat. Aquest bit val zero si el processador es troba en mode sistema, altrament es troba en mode usuari. També es disposa d'un segon bit que indica si es permet el tractament d'excepcions o no. Si aquest bit es troba a zero el processador no realitzarà cap mena de gestió d'excepcions, mentre que si el bit està activat el processador gestionarà les excepcions com s'ha escrit anteriorment.

Aquests dos bits són els bits de menor per del registre d'estat (*figura 4.13*). Aquests, juntament amb els quatre bits de immediatament major pes del registre, formen una pila de parelles de bits.

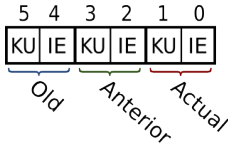


Figura 4.13: Detall dels 6 bits de menor pes del registre d'estat (*Status*). *IE* representa els bits d'habilitació d'excepcions (*Interrupt Enable*) i *KU* els bits de selecció de mode (*Kernel/User*).

Quan es produeix una interrupció, el processador desplaça els sis bits a l'esquerra provocant la pèrdua dels 2 bits de major pes i injectant dos zeros als bits de menor pes (*figura 4.14*). D'aquesta manera es passa a mode sistema amb les excepcions inhibides (ja que *IE*=0 i *KU*=0) però conservant els bits de l'estat anterior.

En el moment d'interpretar una instrucció de retorn d'excepció (*rfe*) el processador realitza el pas invers, desplaça els 6 bits a la dreta retornant a l'estat previ a l'excepció. Els bits de major pes es repliquen i els de menor pes es perden (*figura 4.15*).

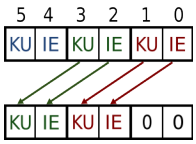


Figura 4.14: Desplaçament del bits d'estat quan es produeix una excepció

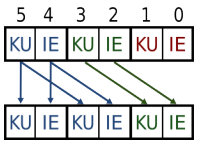


Figura 4.15: Desplaçament dels bits d'estat quan es finalitza una excepció (instrucció *rfe*)

4.4 Rellotge del processador

Tot i que el rellotge es podria considerar un dispositiu (tal com un teclat o una pantalla) per les seves similituds, es tractarà el seu funcionament separadament per ser una part essencial del processador i estar implementada a dins d'aquest.

El rellotge consisteix en un comptador de 32 bits que s'incrementa una unitat a cada cicle de rellotge. Aquest comptador és accessible al programador, tant per a

lectura com modificació, a través del registre número 9 del coprocessador 0. Aquest registre es compara contínuament amb el registre número 11 del coprocessador 0 per a determinar si són iguals. En cas d'igualtat d'aquests s'estableix el bit de *flag* del rellotge (el 8è bit de *flag*) i s'interromp el processador si s'escau.

El programador de sistema pot fer ús d'aquests dos registres per a generar una interrupció al cap d'un cert interval de temps. Només cal establir el valor desitjat als dos registres (comptador i comparador) i permetre les interrupcions de rellotge.

5. Dispositius d'entrada-sortida

Fent ús del hardware que proporciona la placa *DE2-115* s'han implementat tres perifèrics molt útils de cara a l'usuari i de gran utilitat també per a provar el computador i depurar errors. Es detalla el funcionament de cada perifèric així com la seva implementació.

5.1 Pantalla LED de 7 segments

A la placa es troba un conjunt de 8 *LED*s de 7 segments. Cada unitat de 7 segments és capaç de representar una figura apagant i encenent els 7 *LED*s que la formen. S'ha decidit que les 8 unitats representin un únic registre de 32 bits en codificació hexadecimal, ja que es necessiten exactament 8 dígits i a més és fàcil representar en un *display* 7 segments tots els nombres i les sis primeres lletres de l'abecedari.

Un registre de 4 bytes es troba assignat a l'adreça de memòria 0xC0000000. El valor d'aquest registre s'utilitza per il·luminar els *LED*s 7 segments. Tal com es pot veure a la *figura 4.16* la sortida del registre es converteix a símbols representables mitjançant 8 blocs conversors que processen blocs de 4 bits.

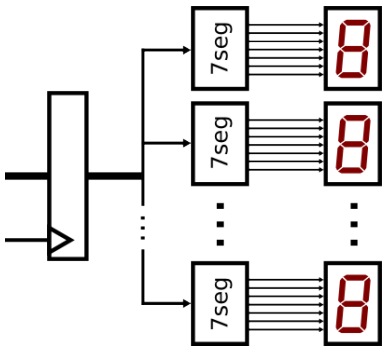


Figura 4.16: Controlador del display de LEDs

5.2 Controlador de pantalla *VGA*

Fent ús de la sortida *VGA* de que disposa la placa de demostració s'ha implementat un dispositiu de pantalla que permet mostrar dades a l'usuari a través del monitor.

Aquest dispositiu proporciona una sortida de vídeo de 640x480 píxels de resolució en blanc i negre i en mode text. Això significa que el processador només pot especificar caràcters de text per la sortida de vídeo.

La representació dels caràcters són símbols de 8 píxels d'amplada i 12 píxels d'alçada. Aquesta mida es tradueix en una pantalla de 80 columnes per 40 línies de caràcters. Es reserva una memòria de 3200 bytes on cada byte representa un caràcter de la pantalla. El processador només ha d'escriure a cada posició de memòria el caràcter, codificat en codi *ASCII*, que vol mostrar i aquest es veurà a la pantalla.

El controlador està format per una memòria per als caràcters amb dos ports d'accés. Un port es fa servir per l'accés del processador i l'altre per l'accés del controlador. Aquest realitza un escombrat a nivell de píxel per a enviar-lo a la pantalla

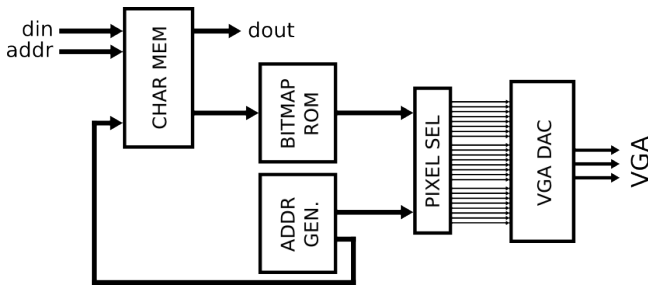


Figura 4.17: Esquema del controlador de pantalla

El controlador genera 24 bits de senyals que corresponen al color d'un píxel. També genera les senyals de sincronisme vertical i horitzontal. Aquestes senyals es transmeten a un *DAC* (Conversor Analògic Digital) que genera senyals analògiques compatibles amb una interfície *VGA*.

5.3 Entrada de teclat *PS/2*

Gràcies al connector *PS/2* situat a la placa es pot connectar un teclat o un ratolí com a perifèric. S'ha optat per implementar un controlador de teclat que permeti llegir les tecles que l'usuari premi.

Cada vegada que l'usuari prem una tecla el controlador escriu en un registre el codi de la tecla i s'activa un bit que indica que el valor de la tecla és vàlid. A més es genera una interrupció al processador per avisar-lo que hi ha una dada vàlida al registre del teclat. Un cop el processador ha llegit el codi de la tecla ha de desactivar el bit de validesa per permetre al controlador llegir més tecles.

A la *figura 4.18* es mostra el registre que implementa el teclat. El registre es troba a l'adreça 0xC0000008.

Figura 4.18: Codificació del registre del teclat

El bit *F* indica si la dada és vàlida. En cas de ser 1 no es llegeixen noves dades del teclat i es genera una interrupció. El processador ha de posar el bit a 0 si desitja adquirir noves dades i evitar la generació de més interrupcions. El codi situat als 8 bits de menor pes del registre (*CODE*) és el número que identifica a la tecla premuda i que depèn de la codificació del teclat. Normalment els teclats fan ús de la mateixa codificació, tot i que hi ha variacions depenent del nombre de tecles i/o l'idioma del teclat.

6. Anàlisi temporal

El software *Quartus* disposa d'una eina de càlcul de camins crítics i retards entre registres [14]. Tot i això, degut a la gran quantitat de registres i connexions que té el processador, és difícil realitzar un anàlisi de *timing* de caràcter aproximat. L'objectiu d'aquest anàlisi és observar quines etapes i entitats tenen un retard major. Conèixer el retard de cada entitat és útil a l'hora de segmentar certes operacions o bé col·locar curtcircuits.
El mètode emprat per a calcular els retards de cada unitat es basa en la síntesi i anàlisi de cada unitat per separat, pel que s'aprecia una diferència entre els resultats obtinguts a tot el processador i en aquestes unitats per separat. Els diferents motius que provoquen aquesta diferència són:

- Algorisme de síntesi no determinista: *Quartus* fa servir un algorisme de col·locació de *LUT*s (en anglès *router*) no determinista. Això comporta que petits canvis en el disseny o en la llavor aleatòria de l'algorisme portin a canvis significatius en els retards del sistema.
- Temps de propagació no menyspreable: Si es sintetitzen unitats per separat el temps de propagació de les senyals és molt petit en comparació amb el temps de propagació del processador complet. La *FPGA* conté busos per a interconnectar els blocs i aquests pateixen congestió si la densitat de blocs és elevada.
- Propagació d'entrada-sortida: Les senyals que requereixen connectar-se a pins d'entrada-sortida pateixen un retard molt més elevat ja que les sortides estan col·locades a grans distàncies. L'algorisme de col·locació pot moure els blocs a llocs propers als pins si ho creu convenient.

A la *figura 4.19* es representen els retards més importants del processador (consultar *annex D* per als valors mesurats). Degut a les aproximacions realitzades, el camí crític no es pot deduir del diagrama. El camí crític en el processador, així com la freqüència de funcionament d'aquest, depenen del nivell d'optimització triat per a l'etapa de síntesi. El camí amb major retard és el que s'inicia al registre entre les etapes *ALU* i *MEM*, passa per la memòria d'usuari (operació de lectura de dades) i, a través del curtcircuit de l'etapa *DL*, arriba al registre on s'emmagatzemen els operands.

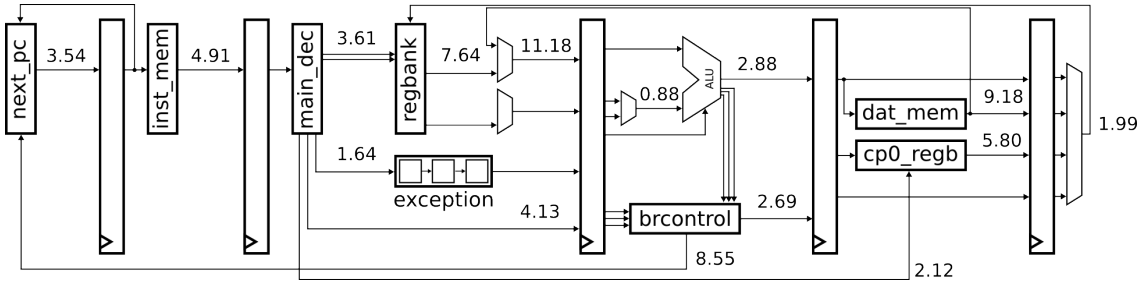


Figura 4.19: Representació dels retards a nivell de bloc al processador (simulació pessimista)

Depenent del nivell d'optimització i del model triat (paràmetres de temperatura, model de simulació, etc.) s'obté una freqüència de treball d'entre 75 i 85MHz. Aquesta dada per sí sola no caracteritza el processador, ja que cal calcular-ne el seu *CPI* (nombre de cicles que es necessiten, en mitjana, per a la interpretació d'una instrucció). En aquest cas el *CPI* serà molt proper a la unitat, ja que hi ha poques instruccions que afegeixin cicles d'espera a la seva interpretació.

Caldria comparar aquests resultats amb els d'un processador no segmentat i els d'un processador segmentat però sense curtcircuits per tal d'esbrinar si és un bon processador o no. La primera comparació la podem realitzar de forma aproximada. Abans d'implementar el processador segmentat es va realitzar una implementació prèvia no segmentada, que tenia una freqüència màxima de treball de 40MHz aproximadament. El seu *CPI* era també molt proper a la unitat (només valia 2 per a les instruccions de lectura de memòria). Es pot veure que la freqüència s'ha doblat pel fet de fer ús de segmentació. A més cal tenir en compte que el processador no segmentat del que es parla no tenia mecanismes d'excepcions.

En definitiva es pot concloure que la segmentació està ben triada, ja que les tres etapes més importants i amb major retard (*CP*, *DL*, *ALU* i *MEM*) tenen retards similars, pel que no cal moure cap entitat d'una etapa a una altra. Les etapes *BUS* i *ES* tenen un retard molt petit, pel que es podria estudiar eliminar-ne alguna part o l'etapa sencera. Si es volgués millorar el temps de cicle es podria dividir l'etapa *DL*

en dues etapes: descodificació i lectura d'operands. Es podria descarregar part del retard de les etapes *ALU* i *DL* a la nova etapa. Lògicament fer això incrementaria el *CPI*, pel que caldria si suposa realment un guany en el rendiment (en termes d'instruccions interpretades per unitat de temps).

Capítol 5

Implementació del nucli

La funció dels computadors és l'execució de tasques dels usuaris d'aquests. Aquestes tasques són programes software consistents en un conjunt d'instruccions i dades que son interpretades per tal d'obtenir resultats d'interès per l'usuari.

Els programes d'ordinador fan ús de dos recursos principalment: la unitat de procés i memòria. La unitat de procés és el recurs que interpreta les instruccions i realitza les operacions que aquestes requereixen mentre que la memòria es fa servir per a guardar les dades i el codi del programa que s'interpreta. A més un programa pot necessitar utilitzar recursos per a comunicar-se amb l'exterior tals com l'usuari, altres programes, etc.

Per a gestionar els recursos d'un computador es van crear els sistemes operatius. Un sistema operatiu és una entitat software que permet als programes d'ordinador l'accés als recursos del computador on s'executen. Entenem per recursos el processador, la memòria i tots els dispositius i perifèrics hardware que pugui tenir un computador.

A grans trets un sistema operatiu s'encarrega de gestionar el processador i la memòria per a repartir-los entre els diferents programes del sistema així com gestionar l'accés als dispositius en favor dels programes.

1. Característiques del nucli

El nucli dissenyat s'adequa, en la mesura del possible, a les recomanacions *POSIX* (*Portable Operating System Interface*, Interfície portable de sistema operatiu) [15]. Això no és sempre possible degut a les limitacions de disseny del processador, ja que aquest no implementa característiques necessàries (o desitjables) per a la gestió de certes funcionalitats del nucli. Principalment el nucli té les responsabilitats de:

- Gestió de processos: Execució i finalització de programes presents al sistema i rotació de processos en execució.
- Entrada i sortida: Accés a perifèrics per a lectura i escriptura de dades.
- Tractament d'excepcions: Captura d'excepcions provocades per programes i tractament d'aquestes.

El sistema permet l'existència d'un determinat nombre de programes que s'ha fixat a 4 en totes les proves realitzades. Aquests programes es troben situats a partir de l'adreça 0 de forma consecutiva i reservant un espai igual per a cada un. També es reserva un espai a la memòria de dades per a aquests programes. A la *figura 5.1* es mostra l'espai de memòria amb detall. Cada programa té un espai de codi de 16KB i 16KB més de dades.

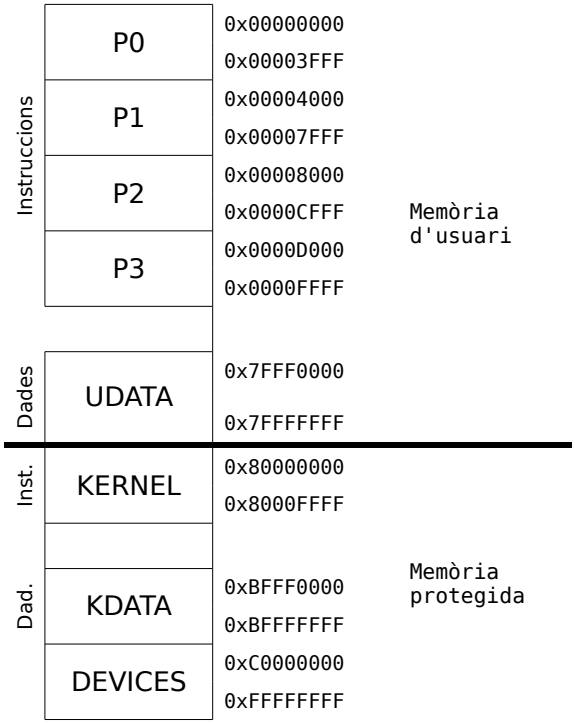


Figura 5.1: Espai de memòria del computador

El codi del nucli es troba dins de l'espai protegit de memòria a partir de l'adreça 0x80000000 de manera que en cas de *reset* o excepció és aquest codi el que rep el flux d'execució. El nucli disposa d'una memòria de dades i d'instruccions de 64KB cada una. Finalment a l'espai protegit de memòria es troben els dispositius d'entrada-sortida.

A l'*annex E* es pot trobar el procediment i les eines necessàries per a construir executables i el nucli del sistema d'acord al que s'ha exposat aquí.

2. Accés al sistema

Tal com es va explicar al capítol d'excepcions i interrupcions el processador treballa normalment en mode no privilegiat executant codi d'usuari. Per accedir al sistema, és a dir, executar codi del nucli en mode privilegiat, cal que es produeixi una excepció. Així doncs les excepcions són l'única manera d'accedir al sistema operatiu un cop aquest es troba interpretant un programa en mode usuari.

2.1 Mecanismes d'accés al sistema

Dividirem totes les possibles excepcions (punts d'entrada al sistema) que es poden produir en tres grans grups. En primer lloc les interrupcions produïdes pels dispositius, en segon lloc les interrupcions del rellotge i finalment les excepcions provocades pel programa de l'usuari.

2.1.1 Interrupcions de dispositius

Quan un dispositiu té dades preparades o vol notificar un esdeveniment que mereix l'atenció del processador aquest provoca una interrupció. Les entrades al sistema provocades per interrupcions de dispositius han de servir per a que el nucli pugui atendre les peticions d'aquests dispositius.

En el cas particular del computador implementat hi ha un dispositiu que és un teclat connectat al sistema mitjançant un controlador PS/2. Aquest controlador genera una interrupció cada vegada que es rep un caràcter del teclat. El nucli és el responsable de llegir el codi de la tecla i guardar-lo a la memòria per a poder rebre més dades i, posteriorment, ser capaç d'entregar les dades als usuaris.

2.1.2 Interrupcions de rellotge

El rellotge és un dispositiu que forma part del processador. Permet l'activació periòdica d'interrupcions de forma controlada pel programador. El nucli fa servir el rellotge per a programar entrades a sistema de forma periòdica. L'objectiu d'aquesta entrada a sistema és realitzar planificació (*scheduling*) de processos.

2.1.3 Excepcions provocades per l'usuari

Els programes d'usuari poden generar diversos tipus d'excepcions. En general aquestes excepcions estan relacionades amb un mal funcionament del programa: accessos a memòria incorrectes, instruccions no vàlides, etc. El nucli és responsable de tractar aquest mal funcionament dels programes d'usuari aturant-ne la seva execució o realitzant l'acció que cregui oportuna.

Per altra banda hi ha certes excepcions provocades per l'usuari que es realitzen de

forma voluntària. Aquestes excepcions es fan servir per a implementar crides a serveis de sistema des d'un programa d'usuari. El processador disposa d'instruccions previstes per aquesta tasca.

2.2 Implementació de l'entrada i la sortida del nucli

Quan es produeix una excepció, el processador activa un mecanisme consistent en inhibir les excepcions i canviar el comptador de programa a una adreça fixada. El nucli del sistema operatiu té una rutina anomenada rutina de servei d'excepcions (o *exception handler*) que està situada a partir d'aquesta adreça i que s'encarrega de tractar l'excepció que s'ha produït.

El procés que realitza aquesta rutina del nucli quan es produeix una excepció és el següent:

1. La rutina guarda el valor de tots els registres de propòsit general i del registre *EPC* (adreça de programa on s'ha produït l'excepció) a la memòria del nucli.
2. La rutina modifica els valors d'alguns registres amb l'objectiu de fer-ne ús. Això inclou registres tals com el punter de pila i l'adreça de retorn.
3. La rutina processa l'excepció. Les accions que es realitzen en aquest punt depenen en gran mesura del tipus d'excepció que s'ha produït.
4. La rutina restaura el valor de tots els registres de propòsit general guardats a la memòria. A continuació passa a mode usuari i estableix el comptador de programa a l'adreça indicada pel *EPC*. Això provoca un retorn al punt on s'havia produït l'excepció de manera transparent (l'usuari no sap que s'ha produït una excepció).

Per a portar a terme el procés de guardar i recuperar l'estat el nucli fa ús dels registres *k0* i *k1*, que son dos registres de propòsit general reservats per al seu ús.

Per a guardar les dades de tots els processos que es troben en execució (valors dels registres així com altres variables d'interès al sistema operatiu) cal disposar d'un espai de memòria per a cada procés en execució. Aquest bloc de memòria s'anomena *PCB* (*Process Control Block*) i es troba a la memòria del sistema operatiu.

3. Crides a sistema

Els programes en execució en el sistema poden demanar al nucli que realitzi certes tasques, tals com la finalització o inici d'un programa, l'escriptura a pantalla, etc. Per a fer aquestes peticions realitzen una crida a sistema (*syscall*). Definirem doncs crida a sistema com un mecanisme d'entrada al nucli iniciat per un programa d'usuari de forma voluntària i amb l'objectiu de demanar al nucli la realització d'alguna tasca.

3.1 Funcionament

Les crides a sistema aprofiten els mecanismes d'excepcions presents al processador, ja que la única manera d'entrar al nucli (mode privilegiat) és mitjançant una excepció. Així doncs els programes d'usuari provoquen excepcions de forma voluntària per a demanar al sistema la realització d'una tasca. Aquest mecanisme funciona de la següent manera:

1. El programa especifica quina tasca vol que es realitzi i els paràmetres necessaris per dur-la a terme. Aquesta informació es manté en els registres del processador.
2. El programa fa ús de la instrucció *syscall* (crida a sistema). En interpretar aquesta instrucció el processador provocarà una excepció que serà gestionada pel nucli.
3. El nucli identifica l'excepció i, fent ús dels paràmetres que ha proporcionat el programa, executa la tasca que aquest ha demanat.
4. El nucli finalitza el tractament de l'excepció retornant al programa d'usuari i continuant-ne l'execució a la instrucció següent a la *syscall*.

A la *figura 5.2* es mostra el codi d'un programa que realitza una crida a sistema.

```
li a0,0      # Primer argument de la crida
li v0,1      # Nombre de crida
syscall      # Instrucció que fa la crida
jr ra
nop
```

Figura 5.2: Programa d'usuari que realitza una crida a sistema

En el cas particular de *POSIX* es fa ús dels registres *a0*, *a1*, *a2* i *a3* per a guardar els paràmetres de la crida i el registre *v0* per a indicar quina crida es vol realitzar. El valor de retorn (si n'hi ha) el col·loca el sistema operatiu al registre *v0*, i a més col·loca el valor 0 o 1 al registre *a3* indicant l'absència o existència d'un error [16].

3.2 Crides implementades

S'han implementat quatre crides a sistema. Aquestes fan ús de la numeració indicada per *POSIX*. A continuació se'n descriu el funcionament.

3.2.1 exec

Descripció: Executa un programa present al sistema.

Paràmetres: 1. Nombre del programa que es desitja executar.

Valor de retorn: Retorna zero en cas d'èxit. En cas de no existir el programa especificat retorna *EINVAL*. Si el programa ja es troba en execució retorna *EEXIST*.

3.2.2 exit

Descripció: Atura l'execució del programa que fa la crida, a excepció del programa número 0, que no es pot aturar.

Paràmetres: No pren cap paràmetre

Valor de retorn: No retorna cap valor ja que el programa no continua la seva execució.

3.2.3 read

Descripció: Llegeix dades d'un dispositiu o fitxer.

Paràmetres: 1. Identificador de la font. El valor zero indica llegir dades del teclat. Qualsevol altre valor no és vàlid.

2. Punter a la posició de memòria on es guardaran les dades llegides. El punter ha de ser vàlid (ha d'estar a l'espai d'adreces del programa que fa la crida).

3. Nombre de *bytes* a llegir.

Valor de retorn: Retorna el nombre de bytes que s'ha llegit en el cas que la lectura hagi estat un èxit. En cas contrari retorna el codi d'error. Aquest pot ser *EINVAL* en cas que la font especificada no sigui vàlida.

3.2.4 write

Descripció: Escriu dades a un dispositiu o fitxer.

Paràmetres: 1. Identificador de la destinació. El valor 1 indica escriure dades a la pantalla i el valor 2 escriure dades al *display LED*. La resta de valors no es consideren vàlids.

2. Punter a la posició de memòria on es troben les dades que es

desitja escriure. El punter ha de ser vàlid (ha d'estar a l'espai d'adreces del programa que fa la crida).

3. Nombre de *bytes* a escriure.

Valor de retorn: Retorna el nombre de bytes que s'han escrit en el cas que la lectura hagi estat un èxit. En cas contrari retorna el codi d'error. Aquest pot ser *EINVAL* en cas que la font especificada no sigui vàlida.

Tota la resta de crides no s'implementen i si l'usuari intenta cridar-les es retorna un error amb codi *ENOSYS*.

4. Planificació de processos

La planificació de processos consisteix canviar periòdicament el procés d'usuari que executa el processador. Realitzant aquesta tasca de forma freqüent s'aconsegueix en l'usuari la sensació que tots els processos s'estan executant simultàniament.

La planificació de processos es realitza quan es produeix una interrupció de rellotge. Aquest ens indica que ha passat una certa quantitat de temps respecte l'última interrupció de rellotge que es va produir. Cada vegada que es produeix aquesta interrupció s'avalua si és necessari canviar de procés i, si ho és, el sistema no retorna al programa que estava executant abans de la interrupció sinó a un altre de diferent. A la *figura 5.3* s'exemplifica aquest procés en un diagrama temporal. Les fletxes representen els instants on es produeix una interrupció de rellotge i es decideix canviar de procés.

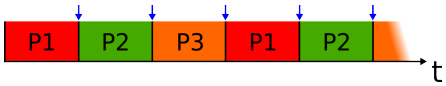


Figura 5.3: Diagrama temporal que exemplifica la rotació de processos.

Per a determinar si cal o no canviar el procés en execució es fa servir un comptador. Cada vegada que es produeix una interrupció de rellotge es decrementa aquest comptador i quan s'arriba a zero es canvia de procés. Aquest comptador rep el nom de *quantum* i s'inicialitza per defecte a un valor que depèn de la freqüència amb la que el rellotge provoca interrupcions.

5. Gestió d'interrupcions

S'entén per gestió d'interrupcions les tasques que porta a terme el sistema quan un perifèric extern al computador genera una interrupció. En el nostre computador només es disposa del teclat com a perifèric capaç de provocar interrupcions. Així doncs només es tractarà la gestió d'aquesta interrupció. El teclat produirà una

interrupció quan arribi un caràcter provinent del teclat pel bus *PS/2*. Aquest caràcter es podrà llegir a través del registre associat a la posició de memòria corresponent (*figura 4.18*).

El flux de tractament de la interrupció serà:

1. Identificar la font de la interrupció mitjançant les *flags* el registre *Cause*.
2. Llegir el caràcter del teclat mitjançant l'adreça de memòria corresponent.
3. Guardar el caràcter (realitzant les conversions oportunes) al *buffer*.
4. Netejar el flag d'interrupció i el registre del teclat.

Un cop realitzades aquestes tasques es pot finalitzar el servei de la interrupció. Quan sigui necessari accedir a les dades (mitjançant la crida *read*, per exemple) aquestes es trobaran al *buffer* i es podrà seguir introduint tecles al teclat.

A la *figura 5.4* es mostra un cronograma corresponent a la simulació del processador duran l'execució d'un codi d'usuari que fa una crida a sistema. La línia vertical groga senyala el canvi de mode d'usuari a sistema.

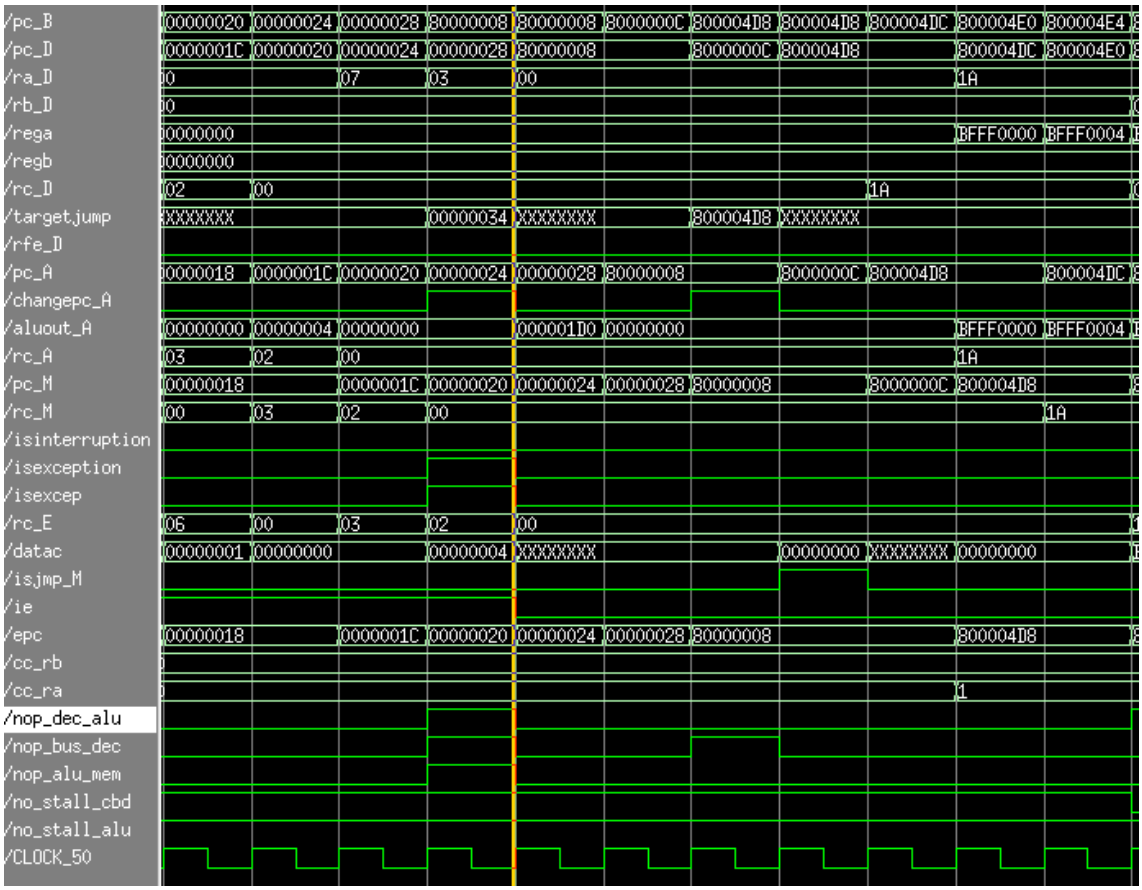


Figura 5.4: Cronograma d'execució d'una crida a sistema

Les etapes es troben ordenades verticalment. Es pot veure el canvi del comptador de programa (*pc_B*, *pc_D*, *pc_A* i *pc_M* depenent de l'etapa) que passa d'una adreça d'usuari 0x28 a una de sistema, concretament l'adreça del *exception handler* 0x80000008. Les senyals d'injecció de *nop*s (*nop_dec_alu*, *nop_bus_dec* i *nop_alu_mem*) s'activen en el cicle d'accés per a descartat totes les instruccions posteriors.

A la *figura 5.5* es mostra la sortida de mode sistema cap a mode usuari. Es fa ús d'una instrucció de salt seguida per una instrucció *rfe*. La línia groga vertical assenyala el canvi de mode, que també es pot veure gràcies a la senyal *rfe_D* la qual indica la interpretació d'una instrucció *rfe*.

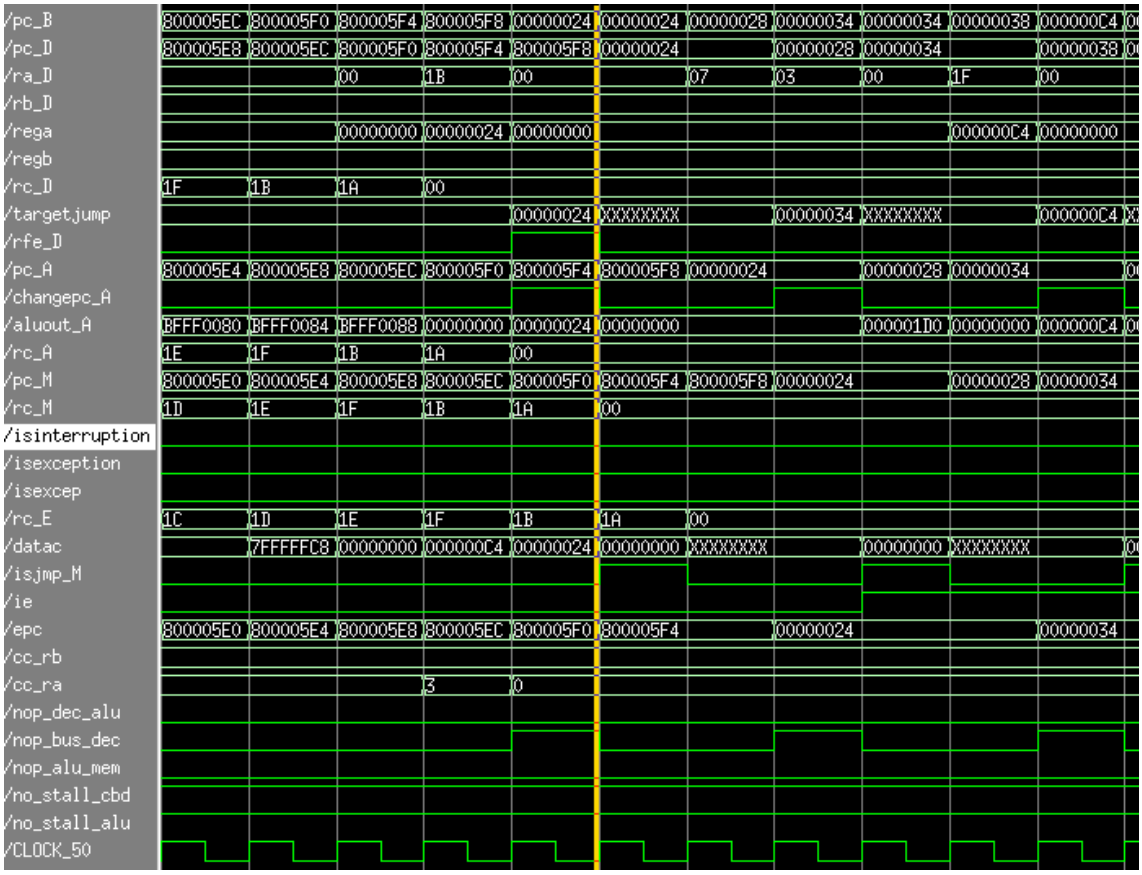


Figura 5.5: Sortida de mode sistema i retorn al codi d'usuari

Capítol 6

Anàlisi econòmic

En aquest capítol es fa un breu anàlisi del cost del projecte. Es té en compte el cost del desenvolupament d'aquest però no s'analitza el cost de les possibles aplicacions d'aquest.

1. Despeses de personal

Dividim les despeses en diferents etapes de desenvolupament. Cada etapa té un cost diferent depenent del grau de complexitat i, per tant, del rol del desenvolupador associat.

	Hores	Preu/Hora	Total (€)
Analista	60	16,28	976,80
Programador	500	14,25	7.125,00
Cap de projecte	20	23,71	474,20
Total	580		8.573,00

2. Despeses de software i hardware

Llistem a continuació un inventari de despeses relacionades amb els elements software i hardware emprats per al desenvolupament del projecte.

Element	Unitats	Preu unitari	Total (€)
Sistema operatiu Linux	1	0	0
Compilador GNU	1	0	0
Altera Quartus II	1	0	0
Placa DE2-115	1	600,00	600,00
Total			600,00

Donat que s'ha utilitzat eines de software lliure el cost del desenvolupament es

limita als recursos hardware emprats. El software *Quartus II* té una llicència gratuïta amb certes limitacions. Donat que no es requereixen les funcionalitats que ofereixen les llicències de pagament amb la versió gratuïta és suficient per a realitzar el projecte.

Capítol 7

Resum

Com a punt i final del projecte es recull a continuació un resum de tot el treball realitzat. Es valorarà tot el que s'ha après durant la realització del mateix i la relació d'aquest amb les assignatures cursades a la carrera.

1. Treball realitzat

Durant el desenvolupament del projecte s'ha:

- Posat en pràctica la metodologia de desenvolupament de sistemes digitals, en particular per al cas d'arquitectures hardware programables.
- Estudiat el llenguatge de programació *VHDL*. Destaquem l'aprenentatge de l'especificació de circuits en alt nivell i les seves tècniques de comprovació.
- Estudiat el funcionament i disseny de l'arquitectura *MIPS*. S'ha realitzat una visió general a les diferents especificacions de l'arquitectura de la família i s'ha treballat en la primera versió d'aquesta (*MIPS I*).
- Profunditzat en el disseny i implementació de processadors, temàtiques apreses en diverses assignatures de l'enginyeria. S'ha reforçat especialment els conceptes de detecció i gestió de riscos així com la predicció de salt.
- S'ha après el funcionament de les *FPGA* i les eines necessàries per a desenvolupar-hi projectes. Destaca l'aprenentatge de les eines de simulació pel seu valor a nivell pràctic.
- S'ha estudiat el funcionament de dispositius tals com les pantalles *VGA* i els teclats *PS/2*, especialment les senyals i protocols dels que fa ús.
- S'han profunditzat els coneixement sobre sistemes operatius i el seu funcionament, especialment en arquitectures diferents a les habituals com la *IA-32*.

2. Objectius assolits

Finalment i després de tot el treball realitzat s'han aconseguit els següents resultats.

- Implementat un processador *MIPS* segmentat amb curtcircuits i detecció de riscos. Aquest processador implementa el joc d'instruccions *MIPS I*.
- Implementats els dispositius controlador de pantalla *VGA* i controlador de teclat *PS/2*. Juntament amb el processador aquests han estat sintetitzats a la *FPGA* de la placa *Terasic DE2-115*. S'ha verificat el funcionament del conjunt.
- S'ha implementat un petit nucli de sistema que realitza les tasques bàsiques: accés als dispositius mitjançant crides a sistema i gestió d'excepcions i d'interrupcions.
- Finalment s'han implementat alguns programes d'usuari per a demostrar el funcionament del sistema. Aquests programes fan ús del nucli i dels dispositius per a interactuar amb l'usuari.

3. Treball futur

El projecte té moltes possibilitats de continuïtat, tant a nivell de disseny digital com a nivell software. A més el projecte és molt portable, pel que es podria continuar el seu desenvolupament en una altra arquitectura.

El desenvolupament del processador es pot continuar implementant una arquitectura de memòria de *Von Neumann* que permeti accedir a memòria d'instruccions i interpretar instruccions de la memòria de dades. El següent pas seria afegir traducció d'adreces fent ús de paginació.

Pel que fa al software es podria implementar un nucli amb més funcionalitats, especialment si es desenvolupa una arquitectura de memòria unificada. Si es disposa de tota la funcionalitat hardware relacionada amb la memòria es podria fins i tot carregar un nucli de sistema com podria ser *Linux*. Això requeriria escriure els controladors necessaris per al nostre hardware.

Annex A

Instruccions implementades

El següent llistat és mostra el conjunt d'instruccions que s'han implementat al processador.

Mnemotècnic	Sintaxi	Descripció	Cicl. Perd.	
Instruccions d'accés a memòria				
LB	`lb $rt, offset($rs)`	Càrrega de byte amb extensió de signe	0	1
LBU	`lbu $rt, offset($rs)`	Càrrega de byte sense extensió de signe		
SB	`sb $rt, offset($rs)`	Escriptura de byte	0	0
LH	`lh $rt, offset($rs)`	Càrrega de half-word amb extensió de signe	0	1
LHU	`lhu $rt, offset($rs)`	Càrrega de haf-word sense extensió de signe		
SH	`sh $rt, offset($rs)`	Escriptura de half-word	0	0
LW	`lw $rt, offset($rs)`	Càrrega de word	0	1
SW	`sw $rt, offset($rs)`	Escriptura de word	0	0
Instruccions de salt				
J	`j target`	Salt incondicional amb literal	1	1
JAL	`jal target`	Crida incondicional amb literal		2
JR	`jr $rs`	Salt incondicional amb registre		1
JALR	`jalr $rs`	Crida incondicional amb registre		2
BEQ	`beq $rs,$rt,imm`	Salt condicional en igualtat de registres	0	1
BNE	`bne $rs,$rt,imm`	Salt condicional en diferència de registres		
BLEZ	`blez $rs,imm`	Salt condicional en menor o igual que zero		
BGTZ	`bgtz $rs,imm`	Salt condicional en major que zero		
BLTZ	`bltz $rs,imm`	Salt condicional en menor que zero		
BGEZ	`bgez $rs,imm`	Salt condicional en major o igual que zero		
Instruccions especials				
SYSCALL	`syscall`	Excepció de crida a sistema	0	1
BREAK	`break`	Excepció de punt d'interrupció		
RFE	`rfe`	Retorn des d'una excepció	0	0
MFC0	`mfc0 $rt,$cpr`	Moviment de registre CP0 a registre general	0	2
MTC0	`mtc0 $rt,$cpr`	Moviment de registre general a registre CP0	0	0

Mnemotècnic	Sintaxi	Descripció	Cicl. Perd.	
Instruccions aritmeticològiques				
ADDI	`addi $rt,$rs,imm`	Suma d'immediat amb overflow	0	0
ADDIU	`addiu $rt,$rs,imm`	Suma d'immediat sense overflow		
SLTI	`slti $rt,$rs,imm`	Comparació amb signe amb literal		
SLTIU	`sltiu $rt,$rs,imm`	Comparació sense signe amb literal		
ANDI	`andi $rt,$rs,imm`	AND bit a bit amb literal		
ORI	`ori $rt,$rs,imm`	OR bit a bit amb literal		
XORI	`xori $rt,$rs,imm`	XOR bit a bit amb literal		
LUI	`lui $rt, imm`	Càrrega de literal a la part superior		
ADD	`add $rd,$rs,$rt`	Suma de registres amb overflow		
ADDU	`addu $rd,$rs,$rt`	Suma de registres sense overflow		
SUB	`sub $rd,$rs,$rt`	Resta de registres amb overflow		
SUBU	`subu $rd,$rs,$rt`	Resta de registres sense overflow		
SLT	`slt $rd,$rs,$rt`	Comparació amb signe de registres		
SLTU	`sltu $rd,$rs,$rt`	Comparació sense signe de registres		
AND	`and $rd,$rs,$rt`	AND bit a bit entre registres		
OR	`or $rd,$rs,$rt`	OR bit a bit entre registres		
XOR	`xor $rd,$rs,$rt`	XOR bit a bit entre registres		
NOR	`nor $rd,$rs,$rt`	NOR bit a bit entre registres		
SLL	`sll $rd,$rt,sa`	Desplaçament lògic a l'esquerra amb literal		
SRL	`srl $rd,$rt,sa`	Desplaçament lògic a la dreta amb literal		
SRA	`sra $rd,$rt,sa`	Desplaçament aritmètic a la dreta amb literal		
SLLV	`sllv $rd,$rt,$rs`	Desplaçament lògic a l'esquerra entre registres		
SRLV	`srlv $rd,$rt,$rs`	Desplaçament lògic a la dreta entre registres		
SRAV	`srav $rd,$rt,$rs`	Desplaçament aritmètic a la dreta entre registres		

Nota: Les dos últimes columnes indiquen el nombre mínim i màxim de cicles perduts per cada instrucció.

Annex B

Implementació VHDL del processador

En aquest annex es realitza una descripció detallada de totes les entitats utilitzades per a la implementació del processador. Es proporciona la seva descripció com a entitat i el codi que descriu el seu comportament. Com a referència es proporciona un esquema jeràrquic d'entitats per a un millor enteniment de les interconnexions entre les entitats.

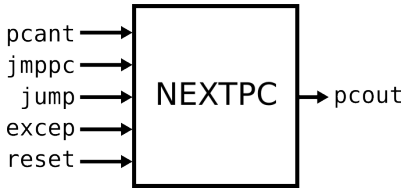
- Unitats del processador segmentat
    1. Comptador de programa
    2. Descodificador
    3. Banc de registres
    4. Unitat de control de salts
    5. Unitat aritmeticològica
    6. Controlador de memòria
        6.1. Descodificador de memòria
        6.2. Controlador d'entrada-sortida
            6.2.1. Controlador de teclat
            6.2.2. Controlador de pantalla
        6.3. Formatador de lectures a memòria
        6.4. Formatador d'escriptures a memòria
    7. Coprocessador 0
    8. Controlador de curtcircuits
    9. Control de riscs
- Unitats auxiliars
    I. Multiplexor
    II. Registre
    III. Propagador d'excepcions

1. Comptador de programa

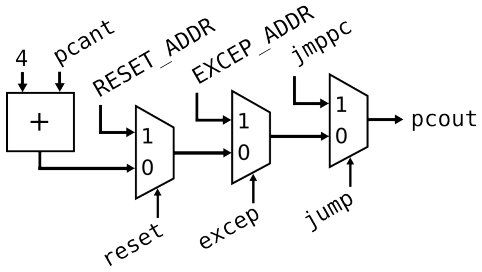
Entitat responsable de la selecció del comptador de programa. Per defecte fa ús de seqüenciament implícit, consistent en sumar 4 (mida de la instrucció en bytes) al *pcant*. En cas que es realitzi un salt (*jump* = 1) s'estableix el nou comptador de programa al valor de *jmppc*. Si hi ha una excepció (*excep* = 1) s'estableix el comptador de programa a 0x80000008 mentre que en el cas de *reset* (*reset* = 1) aquest s'estableix a 0x80000000.

Entitat

```
entity nextpc is port (
  jmppc: in  std_logic_vector(31 downto 0);
  pcout: out std_logic_vector(31 downto 0);
  jump:  in std_logic;
  pcant: in  std_logic_vector(31 downto 0);
  excep: in std_logic;
  reset: in std_logic );
end entity;
```



Arquitectura




```
architecture arch of nextpc is
begin
    process(jmppc,jump,pcant,reset,excep)
    begin
        if (reset = '1') then        pcout <= INSRT_RESET;
        else
            if (excep = '1') then    pcout <= INSRT_EXCEP;
            else
                if (jump = '1') then    pcout <= jmppc;
                else
                    pcout <= std_logic_vector(unsigned(pcant)+4);
                end if;
            end if;
        end if;
    end process;
end architecture;
```

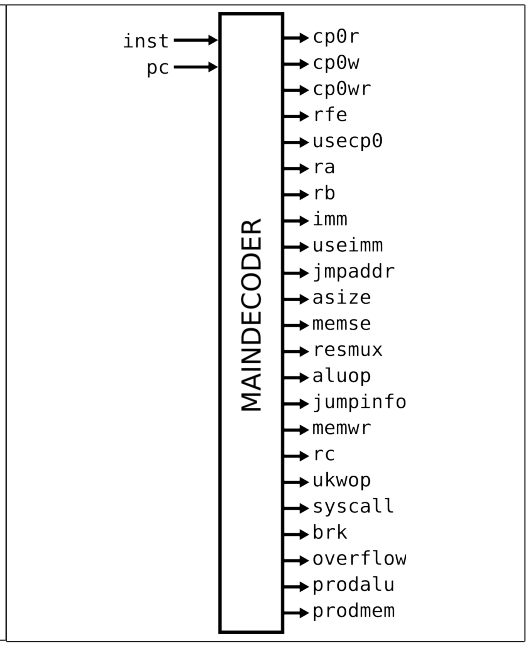
2. Descodificador

Entitat responsable de la generació de les senyals de control del camí de dades. Rep com a entrades la instrucció i l'adreça on es troba aquesta. Les sortides es poden dividir en diferents grups independents.

- Senyals d'accés al banc de registres: Es genera *ra*, *rb* i *rc*, identificadors d'accés al banc de registres per a lectura i escriptura de dades.
- Senyals d'accés a memòria: Mida de l'accés, extensió de signe i permís d'escriptura es codifiquen en *asize*, *memse* i *memwr* respectivament.
- Senyals d'accés al coprocessador: Les senyals *cp0r* i *cp0w* són identificadors de registres del coprocessador i *cp0wr* és el permís d'escriptura en aquest.
- Senyalització d'excepcions: *ukwop*, *syscall*, *brk* i *overflow* són senyals que indiquen l'existència d'una excepció, mentre que *rfe* n'indica la finalització.
- Senyals de control del camí de dades: Indiquen per quins busos circulen les dades: *useimm*, *resmux* i *usecp0*. També donen suport a la lògica de curtcircuits les senyals *prodalu* i *prodmem*.
- Senyals de control de seqüenciament: Es codifica a *jmpinfo* i *jmpaddr* informació necessària per a descodificar instruccions de salt.
- Senyals de la ALU: *aluop* i *imm* codifiquen el codi d'operació de la ALU i l'immediat a utilitzar (si s'escau) respectivament.

Entitat

```
entity maindecoder is port (
  inst:      in std_logic_vector(31 downto 0);
  pc:        in std_logic_vector(31 downto 0);
  cp0r,cp0w:out std_logic_vector( 4 downto 0);
  cp0wr:     out std_logic;
  rfe:       out std_logic;
  usecp0:    out std_logic;
  ra,rb:     out std_logic_vector( 4 downto 0);
  imm:       out std_logic_vector(31 downto 0);
  useimm:    out std_logic;
  jmpaddr:   out std_logic_vector(31 downto 0);
  asize:     out std_logic_vector( 1 downto 0);
  memse:     out std_logic;
  resmux:    out std_logic_vector( 1 downto 0);
  aluop:     out std_logic_vector( 4 downto 0);
  jumpinfo:  out std_logic_vector( 3 downto 0);
  memwr:     out std_logic;
  rc:        out std_logic_vector( 4 downto 0);
  ukwop:     out std_logic;
  syscall:   out std_logic;
  brk:       out std_logic;
  overflow:  out std_logic;
  prodalu:   out std_logic;
  prodmem:   out std_logic );
end entity;
```



Arquitectura

```
architecture arch of maindecoder is
    -- R-Type
    signal opcode: std_logic_vector(5 downto 0);
    signal rs:     std_logic_vector(4 downto 0);
    signal rt:     std_logic_vector(4 downto 0);
    signal rd:     std_logic_vector(4 downto 0);
    signal sa:     std_logic_vector(4 downto 0);
    signal func:   std_logic_vector(5 downto 0);
    -- I-type
    signal immediate: std_logic_vector(15 downto 0);
    -- J-Type
    signal aoffset: std_logic_vector(25 downto 0);
begin
    -- Descodificació dels camps d'instrucció
    opcode <= inst(31 downto 26);
    rs     <= inst(25 downto 21);
    rt     <= inst(20 downto 16);
    rd     <= inst(15 downto 11);
    sa     <= inst(10 downto 6);
    func   <= inst(5 downto 0);
    immediate <= inst(15 downto 0);
    aoffset <= inst(25 downto 0);

    jmpaddr <= pc(31 downto 28) & aoffset & "00";

    -- Càlcul d'indentifadors de registre, immediats i mutiplexor de sortida
    -- Es notifiquen les instruccions mal codificades
    process(opcode,rs,rt,rd,sa,func,immediate,aoffset,inst)
    begin
        case opcode is
        when OP_RTYPE =>
            case func is
            when FUNC_SLL | FUNC_SRL | FUNC_SRA =>
                ra <= rt;
                rb <= "00000";
                rc <= rd;
                useimm <= '1';
                imm <= (31 downto 5 => '-') & sa;
                resmux <= MUX_ALU;
                ukwop <= '0';
            when FUNC_SLLV | FUNC_SRLV | FUNC_SRAV =>
                ra <= rt;
                rb <= rs;
                rc <= rd;
                useimm <= '0';
                imm <= (others => '-');
                resmux <= MUX_ALU;
                ukwop <= '0';
            when FUNC_JR =>
                ra <= rs;
                rb <= "00000";
                rc <= "00000";
                useimm <= '-';
                imm <= (others => '-');
                resmux <= "--";
                ukwop <= '0';
            when FUNC_JALR =>
                ra <= rs;
                rb <= "00000";
                rc <= "11111";
                useimm <= '-';
                imm <= (others => '-');
                resmux <= MUX_PC;
                ukwop <= '0';
            when FUNC_MUL | FUNC_MULU | FUNC_DIV | FUNC_DIVU =>
                ra <= rs;
```

```
            rb <= rt;
            rc <= "00000";
            useimm <= '0';
            imm <= (others => '-');
            resmux <= "--";
            ukwop <= '0';
        when FUNC_MFLO | FUNC_MFHI =>
            ra <= "00000";
            rb <= "00000";
            rc <= rd;
            useimm <= '-';
            imm <= (others => '-');
            resmux <= MUX_ALU;
            ukwop <= '0';
        when FUNC_MTLO | FUNC_MTHI =>
            ra <= rs;
            rb <= "00000";
            rc <= "00000";
            useimm <= '-';
            imm <= (others => '-');
            resmux <= "--";
            ukwop <= '0';
        when FUNC_ADD | FUNC_ADDU | FUNC_SUB | FUNC_SUBU | FUNC_AND |
                FUNC_OR | FUNC_XOR | FUNC_NOR | FUNC_SLT | FUNC_SLTU =>
            ra <= rs;
            rb <= rt;
            rc <= rd;
            useimm <= '0';
            imm <= (others => '-');
            resmux <= MUX_ALU;
            ukwop <= '0';
        when FUNC_SYS | FUNC_BREAK =>
            ra <= "00000";
            rb <= "00000";
            rc <= "00000";
            useimm <= '-';
            imm <= (others => '-');
            resmux <= "--";
            ukwop <= '0';
        when others =>
            ra <= "00000";
            rb <= "00000";
            rc <= "00000";
            useimm <= '-';
            imm <= (others => '-');
            resmux <= "--";
            ukwop <= '1';
        end case;
    when OP_CP0 =>
        case rs is
        when RS_MFC0 =>
            ra <= "00000";
            rb <= "00000";
            rc <= rt;
            useimm <= '-';
            imm <= (others => '-');
            resmux <= MUX_CP0;
            ukwop <= '0';
        when RS_MTC0 =>
            ra <= rt;
            rb <= "00000";
            rc <= "00000";
            useimm <= '-';
            imm <= (others => '-');
            resmux <= "--";
            ukwop <= '0';
        when others =>
            ra <= "00000";
            rb <= "00000";
```

```
            rc <= "00000";
            useimm <= '-';
            imm <= (others => '-');
            resmux <= "--";
            if (inst = INSTRUCTION_RFE) then    ukwop <= '0';
            else                                ukwop <= '1';
            end if;
        end case;
    when OP_ADDI | OP_ADDIU | OP_SLTI | OP_SLTIU =>
        ra <= rs;
        rb <= "00000";
        rc <= rt;
        useimm <= '1';
        imm <= (31 downto 16 => immediate(15)) & immediate;
        resmux <= MUX_ALU;
        ukwop <= '0';
    when OP_ANDI | OP_ORI | OP_XORI =>
        ra <= rs;
        rb <= "00000";
        rc <= rt;
        useimm <= '1';
        imm <= (31 downto 16 => '0') & immediate;
        resmux <= MUX_ALU;
        ukwop <= '0';
    when OP_LUI =>
        ra <= "00000";
        rb <= "00000";
        rc <= rt;
        useimm <= '1';
        imm <= immediate & X"0000";
        resmux <= MUX_ALU;
        ukwop <= '0';
    when OP_JUMP =>
        ra <= "00000";
        rb <= "00000";
        rc <= "00000";
        useimm <= '-';
        imm <= (others => '-');
        resmux <= "--";
        ukwop <= '0';
    when OP_JAL =>
        ra <= "00000";
        rb <= "00000";
        rc <= "11111";
        useimm <= '-';
        imm <= (others => '-');
        resmux <= MUX_PC;
        ukwop <= '0';
    when OP_BEQ | OP_BNE =>
        ra <= rs;
        rb <= rt;
        rc <= "00000";
        useimm <= '0';
        imm <= (31 downto 18 => immediate(15)) & immediate & "00";
        resmux <= "--";
        ukwop <= '0';
    when OP_BRANCH | OP_BLEZ | OP_BGTZ =>
        ra <= rs;
        rb <= "00000";
        if (rt = RT_BLTZAL or rt = RT_BGEZAL) then
            rc <= "11111";
            resmux <= MUX_PC;
        else
            rc <= "00000";
            resmux <= "--";
        end if;
        useimm <= '-';
        imm <= (31 downto 18 => immediate(15)) & immediate & "00";
        ukwop <= '0';
```

```
when OP_LW | OP_LB | OP_LBU | OP_LH | OP_LHU =>
    ra <= rs;
    rb <= "00000";
    rc <= rt;
    useimm <= '1';
    imm <= (31 downto 16 => immediate(15)) & immediate;
    resmux <= MUX_MEM;
    ukwop <= '0';
when OP_SW | OP_SB =>
    ra <= rs;
    rb <= rt;
    rc <= "00000";
    useimm <= '1';
    imm <= (31 downto 16 => immediate(15)) & immediate;
    resmux <= "--";
    ukwop <= '0';
when others =>
    ra <= "00000";
    rb <= "00000";
    rc <= "00000";
    resmux <= "--";
    useimm <= '-';
    imm <= (others => '-');
    ukwop <= '1';
end case;

-- Descodificador de l'ALU
case opcode is
when OP_RTYPE =>
    case func is
    when FUNC_SLL | FUNC_SLLV => aluop <= AOP_SLL;
    when FUNC_SRA | FUNC_SRAV => aluop <= AOP_SRA;
    when FUNC_SRL | FUNC_SRLV => aluop <= AOP_SRL;
    when FUNC_MFHI => aluop <= AOP_HI;
    when FUNC_MFLO => aluop <= AOP_LO;
    when FUNC_MTHI => aluop <= AOP_SETHI;
    when FUNC_MTLO => aluop <= AOP_SETLO;
    when FUNC_MUL  => aluop <= AOP_MUL;
    when FUNC_MULU => aluop <= AOP_MULU;
    when FUNC_ADD | FUNC_ADDU => aluop <= AOP_ADD;
    when FUNC_SUB | FUNC_SUBU => aluop <= AOP_SUB;
    when FUNC_AND  => aluop <= AOP_AND;
    when FUNC_OR   => aluop <= AOP_OR;
    when FUNC_XOR  => aluop <= AOP_XOR;
    when FUNC_NOR  => aluop <= AOP_NOR;
    when FUNC_SLT  => aluop <= AOP_LT;
    when FUNC_SLTU => aluop <= AOP_LTU;
    when others    => aluop <= AOP_A;
    end case;
when OP_CP0 => aluop <= AOP_A;
when OP_ADDI | OP_ADDIU => aluop <= AOP_ADD;
when OP_SLTI  => aluop <= AOP_LT;
when OP_SLTIU => aluop <= AOP_LTU;
when OP_ANDI  => aluop <= AOP_AND;
when OP_ORI   => aluop <= AOP_OR;
when OP_XORI  => aluop <= AOP_XOR;
when OP_LUI   => aluop <= AOP_B;
when OP_LW | OP_LB | OP_LBU | OP_LH | OP_LHU | OP_SW | OP_SB => aluop <=AOP_ADD;
when OP_BRANCH | OP_BLEZ | OP_BGTZ => aluop <= AOP_A;
when others => aluop <= AOP_A;
end case;

-- Descodificador dels salts
case opcode is
when OP_RTYPE =>
    case func is
    when FUNC_JR | FUNC_JALR => jumpinfo <= JUMP_RJUMP;
    when others => jumpinfo <= JUMP_NOJUMP;
    end case;
```

```
when OP_JUMP | OP_JAL => jumpinfo <= JUMP_LJUMP;
when OP_BEQ  => jumpinfo <= JUMP_BEQ;
when OP_BNE  => jumpinfo <= JUMP_BNE;
when OP_BLEZ => jumpinfo <= JUMP_LEZ;
when OP_BGTZ => jumpinfo <= JUMP_GTZ;
when OP_BRANCH =>
    case rt is
    when RT_BGEZAL | RT_BGEZ => jumpinfo <= JUMP_GEZ;
    when RT_BLTZAL | RT_BLTZ => jumpinfo <= JUMP_LTZ;
    when others => jumpinfo <= JUMP_NOJUMP;
    end case;
when others => jumpinfo <= JUMP_NOJUMP;
end case;

-- Descodificador de suport a la lgica de curtcircuis i bloquejos
case opcode is
when OP_RTYPE =>
    case func is
    when    FUNC_SLL | FUNC_SRL | FUNC_SRA | FUNC_SLLV | FUNC_SRLV |
            FUNC_SRAV |
            FUNC_ADD | FUNC_ADDU | FUNC_SUB | FUNC_SUBU | FUNC_AND |
            FUNC_OR | FUNC_XOR | FUNC_NOR | FUNC_SLT | FUNC_SLTU =>

            prodalu <= '1';
            prodmem <= '0';
    when others =>
            prodalu <= '0';
            prodmem <= '0';
    end case;
when     OP_ADDI | OP_ADDIU | OP_SLTI | OP_SLTIU |
        OP_ANDI | OP_ORI | OP_XORI | OP_LUI =>
    prodalu <= '1';
    prodmem <= '0';
when OP_LW | OP_LB | OP_LBU | OP_LH | OP_LHU =>
    prodmem <= '1';
    prodalu <= '0';
when others =>
    prodalu <= '0';
    prodmem <= '0';
end case;

-- Descodificador dels accessos a memòria
case opcode is
when OP_LW =>
    memwr <= '0'; asize <= ACC_WORD;  memse <= '-';
when OP_LB =>
    memwr <= '0'; asize <= ACC_BYTE;  memse <= '1';
when OP_LBU =>
    memwr <= '0'; asize <= ACC_BYTE;  memse <= '0';
when OP_LH =>
    memwr <= '0'; asize <= ACC_HWORD; memse <= '1';
when OP_LHU =>
    memwr <= '0'; asize <= ACC_HWORD; memse <= '0';
when OP_SW =>
    memwr <= '1'; asize <= ACC_WORD;  memse <= '-';
when OP_SB =>
    memwr <= '1'; asize <= ACC_BYTE;  memse <= '-';
when others =>
    memwr <= '0'; asize <= "--";      memse <= '-';
end case;

-- Descodificador del banc de registres del CP0
case opcode is
when OP_CP0 =>
    case rs is
    when RS_MFC0 =>
        cp0r <= rd;
        cp0w <= (others => '-');
        cp0wr <= '0';
```

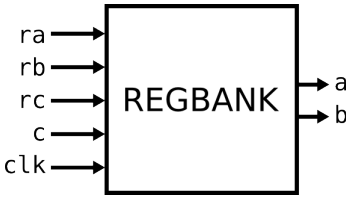
```
                usecp0 <= '1';
            when RS_MTC0 =>
                cp0r <= (others => '-');
                cp0w <= rd;
                cp0wr <= '1';
                usecp0 <= '0';
            when others =>
                cp0r <= (others => '-');
                cp0w <= (others => '-');
                cp0wr <= '0';
                usecp0 <= '0';
            end case;
        when others =>
            cp0r <= (others => '-');
            cp0w <= (others => '-');
            cp0wr <= '0';
            usecp0 <= '0';
        end case;
    end process;

    -- Descodificador d'excepcions
    rfe <= '1' when (inst = INSTRUCTION_RFE) else '0';
    syscall <= '1' when (opcode = OP_RTYPE and func = FUNC_SYS)   else '0';
    brk <=     '1' when (opcode = OP_RTYPE and func = FUNC_BREAK) else '0';
    overflow<= '1' when (opcode = OP_ADDI  and (opcode = OP_RTYPE and
                                        (func = FUNC_ADD or func = FUNC_SUB))) else '0';

end architecture;
```

3. Banc de registres

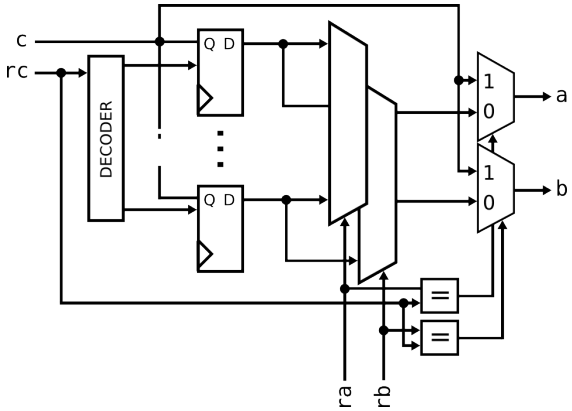
Entitat capaç d'emmagatzemar 32 valors de 32 bits i permetre'n l'accés de lectura i escriptura. Es poden realitzar fins a dues lectures simultàniament a través dels ports *a* i *b* i una escriptura a través del port *c*. L'accés es fa mitjançant un identificador de registre que identifica cada un dels valors emmagatzemats. Cada port, ja sigui de lectura o escriptura, disposa d'una entrada per l'identificador de registre (*ra*, *rb* i *rc*), que és un nombre de 5 bits. El registre número zero sempre conté el valor zero en lectura i les escriptures no tenen efecte.

Entitat

```
entity regbank is port (
  ra,rb,rc: in std_logic_vector(4 downto 0);
  a,b:      out std_logic_vector(31 downto 0);
  c:        in std_logic_vector(31 downto 0);
  clk:      in std_logic );
end entity;
```



Arquitectura




```
architecture arch of regbank is
    type regarray is array(1 to 31) of std_logic_vector(31 downto 0);
    signal registers: regarray;
begin
    process (clk)
    begin
        if (clk'event and clk='1') then
            if (rc > "00000" and rc <= "11111") then
                registers(to_integer(unsigned(rc))) <= c;
            end if;
        end if;
    end process;
    a <=  c                                    when ra = rc and ra /= "00000"      else
          registers(to_integer(unsigned(ra))) when ra > "00000" and ra <= "11111" else
          X"00000000";
    b <=  c                                    when rb = rc and rb /= "00000"      else
          registers(to_integer(unsigned(rb))) when rb > "00000" and rb <= "11111" else
          X"00000000";
end architecture;
```

4. Unitat de control de salts

Entitat responsable de determinar si cal realitzar un salt així com de calcular l'adreça del salt i la de retorn.

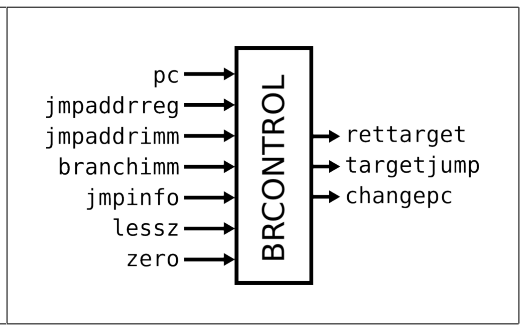
La senyal *jmpinfo* transporta la informació sobre el salt indicant: si és una instrucció que no realitza salt, si és un salt incondicional o bé si és un salt condicional. En aquest últim cas, a més, indica de quin tipus de salt condicional es tracta. Les senyals *lessz, zero* i *equal* es fan servir en el cas dels salts condicionals per avaluar si cal o no realitzar el salt.

Donat que hi ha tres tipus de salts diferents en l'arquitectura, es té una entrada per a cada tipus: *jmpaddrreg* conté el valor del registre per als salts al valor d'un registre, *jmpaddrimm* conté el valor literal de l'adreça per als salts absoluts i *branchimm* conté el desplaçament que cal realitzar en el cas d'un salt relatiu al comptador de programa.

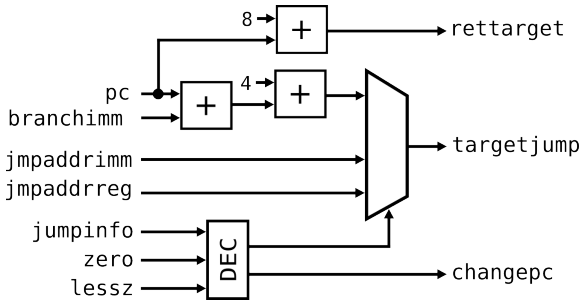
Les sortides són un bit que indica si cal canviar el comptador de programa (*changepc*) i l'adreça a la que cal canviar-lo (*targetjump*). A més es calcula l'adreça de retorn del salt, que es fa servir en cas que el salt sigui una instrucció de crida (*rettarget*).

Entitat

```
entity brcontrol is port (
  pc:          in std_logic_vector(31 downto 0);
  jmpaddrreg:  in std_logic_vector(31 downto 0);
  jmpaddrimm:  in std_logic_vector(31 downto 0);
  branchimm:   in std_logic_vector(31 downto 0);
  jumpinfo:    in std_logic_vector( 3 downto 0);
  zero,lessz:  in std_logic;
  equal:       in std_logic;
  rettarget:  out std_logic_vector(31 downto 0);
  targetjump: out std_logic_vector(31 downto 0);
  changepc:   out std_logic );
end entity;
```



Arquitectura



```
architecture arch of brcontrol is
begin
    rettarget <= std_logic_vector(unsigned(pc)+8);
    process(jmpaddrreg,jmpaddrimm,branchimm,jumpinfo,zero,lessz,equal,pc)
    begin
        case jumpinfo is
        when JUMP_NOJUMP =>
            targetjump <= (others => '-');
            changepc <= '0';
        when JUMP_LJUMP =>
            targetjump <= jmpaddrimm;
            changepc <= '1';
        when JUMP_RJUMP =>
            targetjump <= jmpaddrreg;
            changepc <= '1';
        when JUMP_BEQ =>
            if (equal = '1') then
                changepc <= '1';
                targetjump <= std_logic_vector(unsigned(pc)+unsigned(branchimm)+4);
            else
                changepc <= '0';
                targetjump <= (others => '-');
            end if;
        when JUMP_BNE =>
            if (equal = '0') then
                changepc <= '1';
                targetjump <= std_logic_vector(unsigned(pc)+unsigned(branchimm)+4);
            else
                changepc <= '0';
                targetjump <= (others => '-');
            end if;
        when JUMP_LEZ =>
            if (lessz = '1' or zero = '1') then
                changepc <= '1';
                targetjump <= std_logic_vector(unsigned(pc)+unsigned(branchimm)+4);
            else
                changepc <= '0';
                targetjump <= (others => '-');
            end if;
        when JUMP_GTZ =>
            if (lessz = '0' and zero = '0') then
                changepc <= '1';
                targetjump <= std_logic_vector(unsigned(pc)+unsigned(branchimm)+4);
            else
                changepc <= '0';
                targetjump <= (others => '-');
            end if;
        when JUMP_LTZ =>
            if (lessz = '1' and zero = '0') then
                changepc <= '1';
                targetjump <= std_logic_vector(unsigned(pc)+unsigned(branchimm)+4);
            else
                changepc <= '0';
                targetjump <= (others => '-');
            end if;
        when JUMP_GEZ =>
            if (lessz = '0') then
                changepc <= '1';
                targetjump <= std_logic_vector(unsigned(pc)+unsigned(branchimm)+4);
            else
                changepc <= '0';
                targetjump <= (others => '-');
            end if;
        when others =>
            targetjump <= (others => '-');
            changepc <= '0';
        end case;
    end process;
end architecture;
```

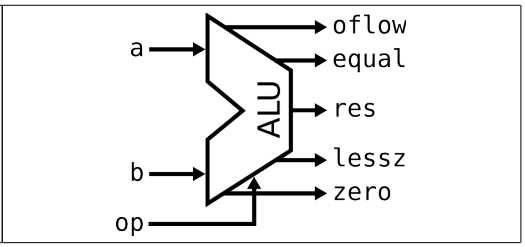
5. Unitat aritmeticològica (*ALU*)

La unitat aritmeticològica (abreujat *ALU*, per les seves sigles anglosaxones) és l'entitat que realitza els càlculs aritmètics i lògics que requereixen les instruccions de càlcul. Els càlculs es realitzen fent ús d'un o dos operands de 32 bits i s'obté un resultat de 32 bits, així com informació addicional sobre l'operació realitzada.

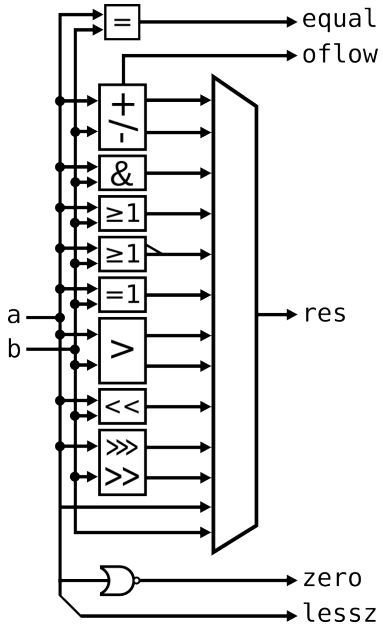
Els operands font fan ús dels ports *a* i *b* per accedir a *l'ALU*. El codi de l'operació a realitzar s'introdueix pel port *op* i s'obté el resultat per la sortida *res*. Es disposa de quatre senyals binàries addicionals que indiquen si el resultat ha provocat un desbordament (*oflow*), si els dos operands són iguals (*equal*) i si l'operand *a* és zero o menor que zero en la seva representació en complement a dos (*zero* i *lessz* respectivament).

Entitat

```
entity alu is port (
  a,b:   in std_logic_vector    (31 downto 0);
  op:    in std_logic_vector    ( 4 downto 0);
  res:   buffer std_logic_vector(31 downto 0);
  oflow: out std_logic;
  lessz: out std_logic;
  zero:  out std_logic;
  equal: out std_logic );
end alu;
```



Arquitectura



```
architecture arch of alu is
    function booleval (cond: boolean) return std_logic_vector is
    begin
        if (cond) then
            return X"00000001";
        else
            return X"00000000";
        end if;
    end function booleval;

begin

    zero <= '1' when (unsigned(a) = 0) else '0';
    lessz <= '1' when (signed(a) < 0) else '0';
    equal <= '1' when (a = b) else '0';

    oflow <= '1' when ( (op = AOP_ADD or op = AOP_SUB) and
                        ( (a(31)='1' and b(31)='1' and res(31)='0') or
                          (a(31)='0' and b(31)='0' and res(31)='1')
                        )
                      )
                else '0';

    res <= std_logic_vector(unsigned(a)+unsigned(b)) when op = AOP_ADD else
           std_logic_vector(unsigned(a)-unsigned(b)) when op = AOP_SUB else

           a and b                                   when op = AOP_AND else
           a or  b                                   when op = AOP_OR  else
           a nor b                                   when op = AOP_NOR else
           a xor b                                   when op = AOP_XOR else

           booleval(signed(a)<signed(b))             when op = AOP_LT  else
           booleval(unsigned(a)<unsigned(b))         when op = AOP_LTU else

           std_logic_vector(shift_right(unsigned(a),
              to_integer(unsigned(b(4 downto 0)))))  when op = AOP_SRL else
           std_logic_vector(shift_right(signed(a),
              to_integer(unsigned(b(4 downto 0)))))  when op = AOP_SRA else
           std_logic_vector(shift_left(unsigned(a),
              to_integer(unsigned(b(4 downto 0)))))  when op = AOP_SLL else

           b                                         when op = AOP_B else
           a                                         when op = AOP_A else
           (others => '-');

end architecture;
```

6. Controlador de memòria

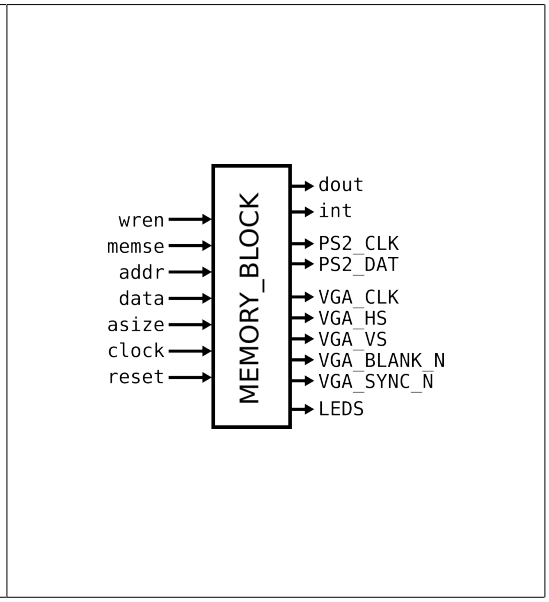
El controlador de memòria és l'entitat que agrupa tots els elements accessibles mitjançant adreces de memòria. Això inclou la memòria de dades però també els dispositius accessibles a través del controlador d'entrada-sortida.

La interfície és l'habitual en el cas d'una memòria: un bus d'adreces (*addr*) i un bus de dades d'entrada (*data*), un bus de dades de sortida (*dout*) i un bit de permís d'escriptura (*wren*). Donat que el bus és de 32 bits i cal permetre accessos a nivell de byte i *halfword* es disposa de les senyals *asize*, per la mida de l'accés i *memse*, per a realitzar extensió de signe a la sortida.

La resta de busos de comunicació són cables que interconnecten els diferents controladors de dispositius amb els connectors de la FPGA. Per restriccions del programa (*Quartus II*) es requereix que aquests cables estiguin a l'entitat superior de la jerarquia.

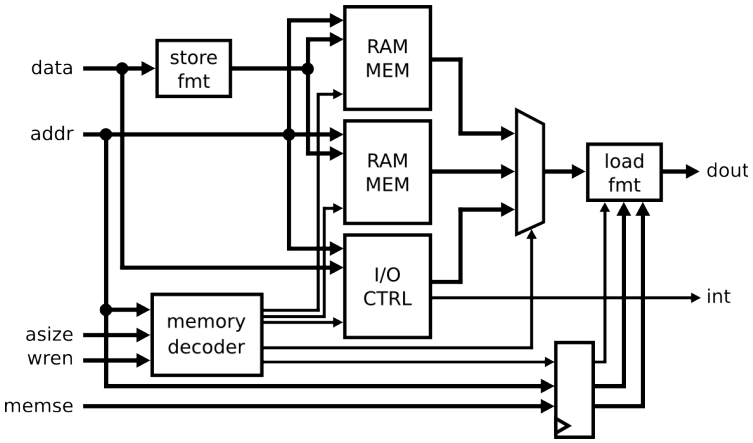
Entitat

```
entity memory_block is port (
  wren:     in std_logic;
  clock:    in std_logic;
  memse:    in std_logic;
  reset:    in std_logic;
  PS2_CLK:  inout std_logic;
  PS2_DAT:  inout std_logic;
  addr:     in std_logic_vector(31 downto 0);
  asize:    in std_logic_vector(1 downto 0);
  data:     in std_logic_vector(31 downto 0);
  SW:       in std_logic_vector(15 downto 0);
  VGA_CLK:  out std_logic;
  VGA_HS:   out std_logic;
  VGA_VS:   out std_logic;
  VGA_BLANK_N: out  std_logic;
  VGA_SYNC_N:  out  std_logic;
  int:      out std_logic;
  dout:     out std_logic_vector(31 downto 0);
  LEDS:     out std_logic_vector(55 downto 0);
  VGA_B:    out std_logic_vector( 7 downto 0);
  VGA_G:    out std_logic_vector( 7 downto 0);
  VGA_R:    out std_logic_vector(7 downto 0));
end memory_block;
```



Arquitectura

L'arquitectura associada a aquesta entitat s'ha descrit fent ús de les eines del programa *Quartus II* de manera que la seva descripció consisteix en un diagrama de blocs enlloc d'una implementació *VHDL*. El codi *VHDL* ha estat, per tant, generat de manera automàtica pel compilador.




```
COMPONENT g4_register
GENERIC (width1 : INTEGER; width2 : INTEGER; width3 : INTEGER; width4 : INTEGER);
    PORT(clk : IN STD_LOGIC;
         enable : IN STD_LOGIC;
         reset : IN STD_LOGIC;
         D1 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
         D2 : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
         D3 : IN STD_LOGIC_VECTOR(0 TO 0);
         D4 : IN STD_LOGIC_VECTOR(0 TO 0);
         Q1 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
         Q2 : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
         Q3 : OUT STD_LOGIC_VECTOR(0 TO 0);
         Q4 : OUT STD_LOGIC_VECTOR(0 TO 0) );
END COMPONENT;

COMPONENT main_mem
    PORT(wren : IN STD_LOGIC;
         clock : IN STD_LOGIC;
         address : IN STD_LOGIC_VECTOR(13 DOWNTO 0);
         byteena : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
         data : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         q : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) );
END COMPONENT;

COMPONENT storefmt
    PORT(addr : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
         asize : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
         din : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         dout : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) );
END COMPONENT;

COMPONENT main_mem_user
    PORT(wren : IN STD_LOGIC;
         clock : IN STD_LOGIC;
         address : IN STD_LOGIC_VECTOR(13 DOWNTO 0);
         byteena : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
         data : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         q : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) );
END COMPONENT;

COMPONENT mux32_3
    PORT(a : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         b : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         c : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         sel : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
         o : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) );
END COMPONENT;

SIGNAL addr_n :  STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL asize_n :  STD_LOGIC_VECTOR(1 DOWNTO 0);
```

```
SIGNAL datafmt :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL dev_wren :  STD_LOGIC;
SIGNAL kram_wren :  STD_LOGIC;
SIGNAL memse_n :  STD_LOGIC;
SIGNAL needfmt :  STD_LOGIC;
SIGNAL ram_ben :  STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL swi :  STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL uram_wren :  STD_LOGIC;
SIGNAL SYNTHESIZED_WIRE_0 :  STD_LOGIC;
SIGNAL SYNTHESIZED_WIRE_1 :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL SYNTHESIZED_WIRE_2 :  STD_LOGIC;
SIGNAL SYNTHESIZED_WIRE_3 :  STD_LOGIC;
SIGNAL SYNTHESIZED_WIRE_4 :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL SYNTHESIZED_WIRE_5 :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL SYNTHESIZED_WIRE_6 :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL SYNTHESIZED_WIRE_7 :  STD_LOGIC_VECTOR(1 DOWNTO 0);

BEGIN

SYNTHESIZED_WIRE_2 <= '1';
SYNTHESIZED_WIRE_3 <= '0';

b2v_inst : loadfmt
PORT MAP(memse => memse_n,
         fmt => SYNTHESIZED_WIRE_0,
         addr => addr_n,
         asize => asize_n,
         din => SYNTHESIZED_WIRE_1,
         dout => dout);


b2v_inst1 : map_io
PORT MAP(clk => clock,
         wren => dev_wren,
         reset => reset,
         PS2_CLK => PS2_CLK,
         PS2_DAT => PS2_DAT,
         addr => addr(29 DOWNTO 0),
         din => data,
         switches => swi,
         int => int,
         VGA_CLK => VGA_CLK,
         VGA_VS => VGA_VS,
         VGA_HS => VGA_HS,
         VGA_BLANK_N => VGA_BLANK_N,
         VGA_SYNC_N => VGA_SYNC_N,
         dout => SYNTHESIZED_WIRE_6,
         leds => LEDS,
         VGA_B => VGA_B,
         VGA_G => VGA_G,
         VGA_R => VGA_R);

b2v_inst2 : memory_decoder
PORT MAP(wren => wren,
         clk => clock,
         addr => addr,
         asize => asize,
         uram_wen => uram_wren,
         kram_wen => kram_wren,
         dev_wen => dev_wren,
         ram_ben => ram_ben,
         sel => SYNTHESIZED_WIRE_7);
b2v_inst3 : g4_register
GENERIC MAP(width1 => 2, width2 => 2, width3 => 1, width4 => 1)
PORT MAP(clk => clock,
         enable => SYNTHESIZED_WIRE_2,
         reset => SYNTHESIZED_WIRE_3,
         D1 => addr(1 DOWNTO 0),
         D2 => asize,
```

```
         D3(0) => memse,
         D4(0) => needfmt,
         Q1 => addr_n,
         Q2 => asize_n,
         Q3(0) => memse_n,
         Q4(0) => SYNTHESIZED_WIRE_0);

b2v_inst44 : main_mem
PORT MAP(wren => kram_wren,
         clock => clock,
         address => addr(15 DOWNTO 2),
         byteena => ram_ben,
         data => datafmt,
         q => SYNTHESIZED_WIRE_5);

b2v_inst5 : storefmt
PORT MAP(addr => addr(1 DOWNTO 0),
         asize => asize,
         din => data,
         dout => datafmt);

b2v_inst6 : main_mem_user
PORT MAP(wren => uram_wren,
         clock => clock,
         address => addr(15 DOWNTO 2),
         byteena => ram_ben,
         data => datafmt,
         q => SYNTHESIZED_WIRE_4);

b2v_inst7 : mux32_3
PORT MAP(a => SYNTHESIZED_WIRE_4,
         b => SYNTHESIZED_WIRE_5,
         c => SYNTHESIZED_WIRE_6,
         sel => SYNTHESIZED_WIRE_7,
         o => SYNTHESIZED_WIRE_1);

swi <= SW;

END bdf_type;
```

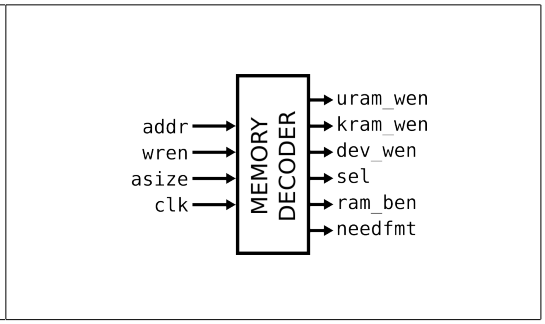
6.1 Descodificador de memòria

El descodificador de memòria és la unitat encarregada de controlar els accessos a memòria per part del processador. Té la responsabilitat d'arbitrar les lectures i escriptures entre els diferents blocs de memòria (i dispositius). La seva implementació determina l'arranjament de memòria, ja que assigna les adreces virtuals a adreces físiques.

Donada una adreça (*addr*), tipus d'accés (*wren*) i mida de l'accés (*asize*) el descodificador calcula a quin bloc cal accedir ja sigui per lectura (senyal de selecció *sel*) o escriptura (permisos d'escriptura *uram_wen*, *kram_wen*, *dev_wen*). A més per als accessos a nivell de byte o halfword calcula la màscara d'escriptura (*ram_ben*).

Entitat

```
entity memory_decoder is port (
  addr:     in  std_logic_vector(31 downto 0);
  wren:     in  std_logic;
  asize:    in  std_logic_vector(1 downto 0);
  clk:      in  std_logic;
  uram_wen:out std_logic;
  kram_wen:out std_logic;
  dev_wen: out std_logic;
  sel:      out std_logic_vector(1 downto 0);
  ram_ben: out std_logic_vector(3 downto 0);
  needfmt: out std_logic);
end entity;
```



Arquitectura

```
architecture arch of memory_decoder is
    signal addr_i: std_logic_vector(31 downto 0);
begin
    uram_wen <= wren when (addr(31 downto 16) = X"7FFF") else '0';
    kram_wen <= wren when (addr(31 downto 16) = X"BFFF") else '0';
    dev_wen  <= wren when (addr(31 downto 16) = X"C000") else '0';

    sel <= "00" when (addr_i(31 downto 16) = X"7FFF") else
           "01" when (addr_i(31 downto 16) = X"BFFF") else
           "10";

    needfmt <= '0' when (addr(31 downto 16) = X"C000") else '1';

    ram_ben <= "1111"   when (asize = ACC_WORD) else
               "0011"   when (asize = ACC_HWORD and addr(1) = '0') else
               "1100"   when (asize = ACC_HWORD and addr(1) = '1') else
               "0001"   when (asize = ACC_BYTE and addr(1 downto 0) = "00") else
               "0010"   when (asize = ACC_BYTE and addr(1 downto 0) = "01") else
               "0100"   when (asize = ACC_BYTE and addr(1 downto 0) = "10") else
               "1000";

    process (clk)
    begin
        if (clk'event and clk = '1') then
            addr_i <= addr;
        end if;
    end process;
end architecture;
```

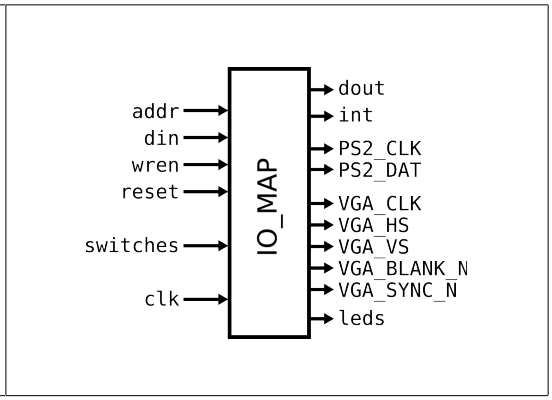
6.2 Controlador d'entrada-sortida

El controlador d'entrada-sortida és una entitat que, fent ús d'una interfície similar a una memòria, permet l'accés als dispositius implementats. A dins hi trobem quatre controladors: pantalla, teclat, *display LED* i interruptors. Els controladors del *display* i dels interruptors estan implementats directament en el controlador d'entrada-sortida, mentre que els controladors de pantalla i teclat estan encapsulats degut al seu grau de complexitat.

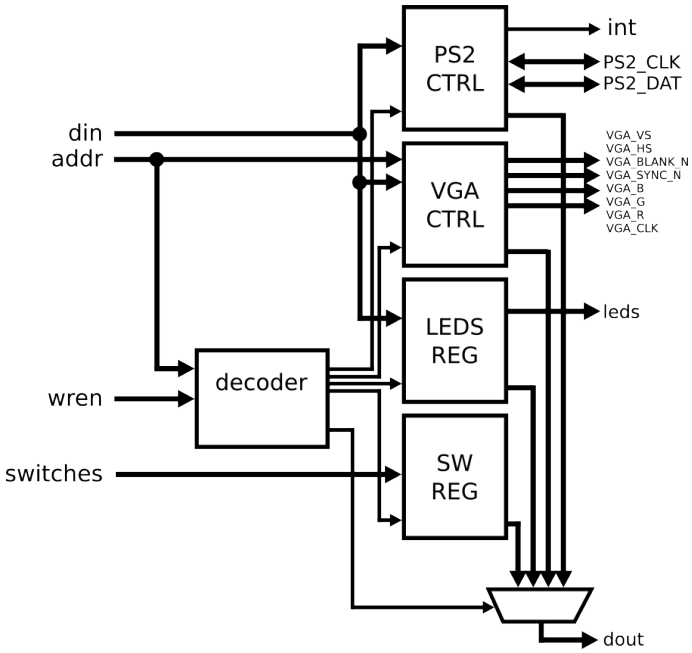
Les senyals *addr*, *din*, *dout* i *wren* proporcionen la interfície de memòria. La resta de senyals representen les sortides de la FPGA que van directament connectades al hardware de la placa.

Entitat

```
entity map_io is port (
  addr:    in std_logic_vector(29 downto 0);
  din :    in std_logic_vector(31 downto 0);
  clk,wren:in std_logic;
  reset:   in std_logic;
  dout:   out std_logic_vector(31 downto 0);
  int:    out std_logic;
  switches:in std_logic_vector(15 downto 0);
  leds:   out std_logic_vector(55 downto 0);
  VGA_CLK,VGA_VS,VGA_HS:out std_logic;
  VGA_BLANK_N,VGA_SYNC_N:out std_logic;
  VGA_B,VGA_G,VGA_R:out
              std_logic_vector(7 downto 0);
  PS2_CLK, PS2_DAT:inout std_logic);
end entity;
```



Arquitectura



```
architecture arch of map_io is
    signal hex_value: std_logic_vector(31 downto 0);
    signal addr_i: std_logic_vector(29 downto 0);
    signal din_i: std_logic_vector(31 downto 0);
    signal wren_i: std_logic;
    signal screen_wen : std_logic;
    signal screenout: std_logic_vector(7 downto 0);
    signal keybd_wen : std_logic;
    signal keybdout: std_logic_vector(31 downto 0);

    function hex7seg (v: std_logic_vector(3 downto 0)) return std_logic_vector is
    begin
        if    (v = X"0") then return "1000000";
        elsif (v = X"1") then return "1111001";
        elsif (v = X"2") then return "0100100";
        elsif (v = X"3") then return "0110000";
        elsif (v = X"4") then return "0011001";
        elsif (v = X"5") then return "0010010";
        elsif (v = X"6") then return "0000010";
        elsif (v = X"7") then return "1111000";
        elsif (v = X"8") then return "0000000";
        elsif (v = X"9") then return "0011000";
        elsif (v = X"A") then return "0001000";
        elsif (v = X"B") then return "0000011";
        elsif (v = X"C") then return "1000110";
        elsif (v = X"D") then return "0100001";
        elsif (v = X"E") then return "0000110";
        else                  return "0001110";
        end if;
    end function hex7seg;

    component vga_driver is port (
            reset :  IN  STD_LOGIC;
            clk :  IN  STD_LOGIC;
            wen :  IN  STD_LOGIC;
            addrin :  IN  STD_LOGIC_VECTOR(11 DOWNTO 0);
            din :  IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
            VGA_CLK :  OUT  STD_LOGIC;
            VGA_HS :  OUT  STD_LOGIC;
            VGA_VS :  OUT  STD_LOGIC;
            VGA_BLANK_N :  OUT  STD_LOGIC;
            VGA_SYNC_N :  OUT  STD_LOGIC;
            dout :  OUT  STD_LOGIC_VECTOR(7 DOWNTO 0);
            VGA_B :  OUT  STD_LOGIC_VECTOR(7 DOWNTO 0);
            VGA_G :  OUT  STD_LOGIC_VECTOR(7 DOWNTO 0);
            VGA_R :  OUT  STD_LOGIC_VECTOR(7 DOWNTO 0) );
    end component;

    component ps2_keyboard is port (
            ps2_clk:  inout    std_logic;
            ps2_dat:  inout    std_logic;

            int:         out std_logic;
            datao:    out std_logic_vector(31 downto 0);
            datai:    in  std_logic_vector(31 downto 0);
            wen:         in  std_logic;

            reset, clk:in std_logic );
    end component;

begin

    -- Output
    dout <= hex_value                    when (unsigned(addr_i) = 0) else
            X"0000" & switches           when (unsigned(addr_i) = 4) else
            keybdout                     when (unsigned(addr_i) = 8) else
            (31 downto 8 =>'-')&screenout when (unsigned(addr_i(29 downto 12))=1) else
            X"DEADBEEF";
```

```
    -- Write process
    process(clk)
    begin
        if (clk'event and clk = '1') then
            addr_i <= addr;
            wren_i <= wren;
            din_i <= din;
            if (wren_i = '1') then
                if (unsigned(addr_i) = 0) then
                    hex_value <= din_i;
                end if;
            end if;
        end if;
    end process;

    -- Screen
    screen_wen <= '1' when (wren = '1' and unsigned(addr(29 downto 12)) = 1) else '0';

    screen_dev: vga_driver PORT MAP (reset => reset, clk => clk, wen => screen_wen,
                                     addrin => addr(11 downto 0), din => din(7 downto 0),
                                     dout => screenout,
                                     VGA_CLK => VGA_CLK, VGA_VS => VGA_VS,
                                     VGA_HS => VGA_HS, VGA_BLANK_N => VGA_BLANK_N,
                                     VGA_SYNC_N => VGA_SYNC_N, VGA_B => VGA_B,
                                     VGA_G => VGA_G, VGA_R => VGA_R );

    -- Keyboard
    keybd_wen <= '1' when (wren = '1' and unsigned(addr) = 8) else '0';

    keyboard_dev: ps2_keyboard PORT MAP (reset => reset, clk => clk,
                                         datao =>keybdout, datai =>din, wen =>keybd_wen,
                                         ps2_clk => PS2_CLK, ps2_dat => PS2_DAT,
                                         int => int);

    -- LED output
    leds <=  hex7seg(hex_value(31 downto 28)) & hex7seg(hex_value(27 downto 24)) &
             hex7seg(hex_value(23 downto 20)) & hex7seg(hex_value(19 downto 16)) &
             hex7seg(hex_value(15 downto 12)) & hex7seg(hex_value(11 downto 8 )) &
             hex7seg(hex_value(7  downto  4)) & hex7seg(hex_value(3  downto 0 ));
end arch;
```

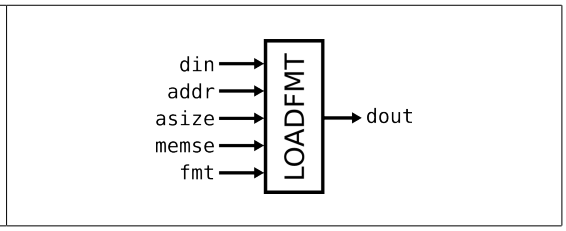
6.3 Formatador de lectures a memòria

Unitat encarregada de tractar les dades llegides a memòria per tal de donar el format adequat a aquestes. En el cas de lectures a nivell de *byte* o *halfword* i depenent dels bits de menor pes de l'adreça la dada llegida es pot trobat a diferents posicions del bus de dades. Aquesta unitat corregeix la posició en el bus i tracta els bits de major pes de l'accés.

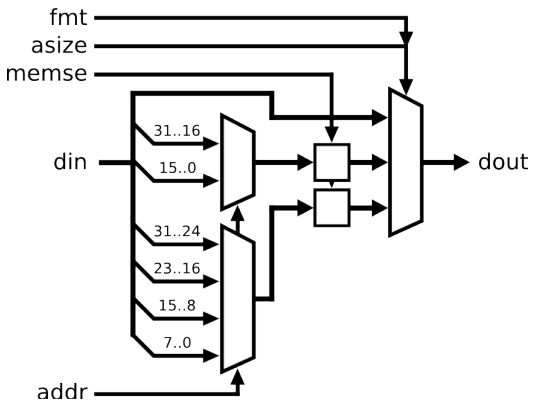
Les dades es llegeixen pel bus *din* i surten pel bus *dout*. Fent ús de l'adreça (*addr*), la mida de l'accés (*asize*) i el bit d'extensió de signe (*memse*) es dóna format a la dada. En cas que *fmt* valgui zero no es dóna format. Aquesta senyal és d'utilitat ens els accessos a dispositius, ja que aquests tenen busos de mides diferents i no requereixen desplaçament de bits.

Entitat

```
entity loadfmt is port (
  din:  in  std_logic_vector(31 downto 0);
  addr: in  std_logic_vector( 1 downto 0);
  asize:in  std_logic_vector( 1 downto 0);
  memse:in  std_logic;
  fmt:  in  std_logic;
  dout: out std_logic_vector(31 downto 0));
end entity;
```



Arquitectura



```
architecture arch of loadfmt is
begin
   process (din,asize,addr,memse,fmt)
   begin
      if (fmt = '1') then
         case asize is
         when ACC_WORD => dout <= din;
         when ACC_HWORD =>
            if (memse = '1') then
               case addr(1) is
               when ('0')  => dout <= (31 downto 16 => din(15)) & din(15 downto 0);
               when others => dout <= (31 downto 16 => din(31)) & din(31 downto 16);
               end case;
            else
               case addr(1) is
               when ('0')  => dout <= (31 downto 16 => '0') & din(15 downto 0);
               when others => dout <= (31 downto 16 => '0') & din(31 downto 16);
               end case;
            end if;
         when others =>
            if (memse = '1') then
               case addr(1 downto 0) is
               when "00" =>   dout <= (31 downto 8 => din( 7)) & din( 7 downto  0);
               when "01" =>   dout <= (31 downto 8 => din(15)) & din(15 downto  8);
               when "10" =>   dout <= (31 downto 8 => din(23)) & din(23 downto 16);
               when others => dout <= (31 downto 8 => din(31)) & din(31 downto 24);
               end case;
            else
               case addr(1 downto 0) is
               when "00" =>   dout <= (31 downto 8 => '0') & din( 7 downto  0);
               when "01" =>   dout <= (31 downto 8 => '0') & din(15 downto  8);
               when "10" =>   dout <= (31 downto 8 => '0') & din(23 downto 16);
               when others => dout <= (31 downto 8 => '0') & din(31 downto 24);
               end case;
            end if;
         end case;
      else
         dout <= din;
      end if;
   end process;

end architecture;
```

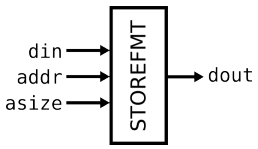
6.4 Formatador d'escriptures a memòria

Unitat encarregada de tractar les dades que s'escriuran a memòria per tal de donar-los el format adequat al tipus d'accés. Donat que els accessos a dades menors d'una paraula (*word*) poden escriure's a diferents posicions d'aquesta l'entitat s'encarrega de desplaçar els bits adequadament.

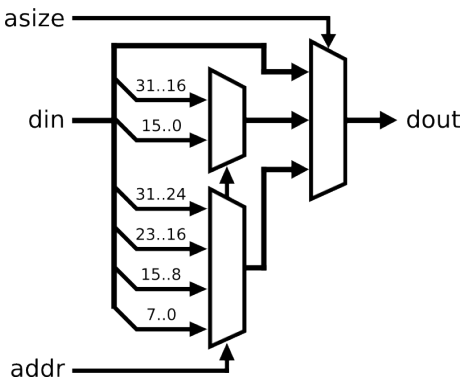
Les dades es llegeixen pel bus *din* i surten pel bus *dout*. Fent ús de l'adreça (*addr*) i la mida de l'accés (*asize*) es col·loca la dada a la part del bus corresponent.

Entitat

```
entity storefmt is port (
  din:  in  std_logic_vector(31 downto 0);
  addr: in  std_logic_vector( 1 downto 0);
  asize:in  std_logic_vector( 1 downto 0);
  dout: out std_logic_vector(31 downto 0));
end entity;
```



Arquitectura




```
architecture arch of storefmt is
begin
  process (din,asize,addr)
  begin
    case asize is
    when ACC_WORD => dout <= din;
    when ACC_HWORD =>
      case addr(1) is
      when ('0')  =>   dout <= (31 downto 16 => '-') & din(15 downto 0);
      when others => dout <= din(15 downto 0) & (15 downto 0 => '-');
      end case;
    when others =>
      case addr(1 downto 0) is
      when "00" => dout<= (31 downto 8 => '-') & din(7 downto 0);
      when "01" => dout<= (31 downto 16 => '-')& din(7 downto 0) & (7 downto 0 => '-');
      when "10" => dout<= (31 downto 24 => '-')& din(7 downto 0) & (15 downto 0 => '-');
      when others => dout <= din(7 downto 0) & (23 downto 0 => '-');
      end case;
    end case;
  end process;
end architecture;
```

7. Coprocessador 0

El coprocessador 0 és una entitat que implementa la major part de les funcionalitats requerides per les excepcions: banc de registres, rellotge de sistema, sistema d'interrupcions i sistema d'excepcions.

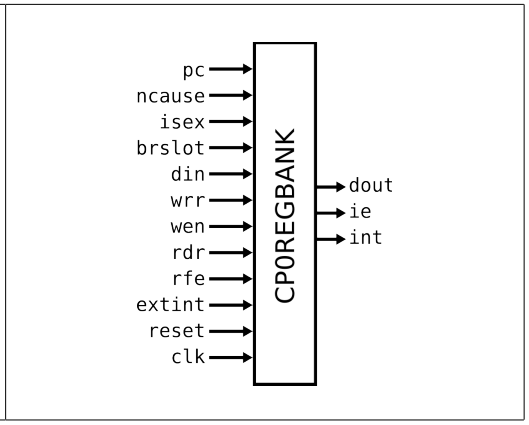
Es poden diferenciar els quatre blocs esmentats. En primer lloc el banc de registres consta d'un port de lectura amb les senyals *rdr* i *dout* (identificador de registre i dada respectivament) i d'un port d'escriptura amb les senyals *wrr*, *wen* i *din* (identificador, permís d'escriptura i dada). Qualsevol escriptura al banc de registres en un cicle on es produeix una excepció no té efecte.

El rellotge fa ús dels registres del banc i no té cap senyal d'entrada-sortida associada. El sistema d'interrupcions disposa de la senyal *extint* que indica l'existència d'una interrupció per part d'un perifèric.

El sistema d'excepcions fa ús de tota la resta de senyals. Necessita l'adreça de la instrucció actual (*pc*) així com saber si és un salt retardat (*brslot*). El codi de l'excepció s'indica a l'entrada *ncause* i el bit que indica si hi ha una excepció és *isex*. La senyal *rfe* indica si cal retornar d'una excepció. Les sortides *ie* i *int* indiquen si les interrupcions estan permeses i si hi ha una excepció o interrupció pendent respectivament.

Entitat

```
entity cp0regbank is port (
  pc:       in std_logic_vector(31 downto 0);
  ncause:   in std_logic_vector( 4 downto 0);
  isex:     in std_logic;
  brslot:   in std_logic;
  din:      in std_logic_vector(31 downto 0);
  wrr:      in std_logic_vector( 4 downto 0);
  wen:      in std_logic;
  rdr:      in std_logic_vector( 4 downto 0);
  dout:    out std_logic_vector(31 downto 0);
  ie:   buffer std_logic;
  rfe:      in std_logic;
  int:      buffer std_logic;
  extint:   in std_logic;
  reset,clk:in std_logic);
end entity;
```



Arquitectura

```
architecture arch of cp0regbank is
    signal epc:     std_logic_vector(31 downto 0);  -- EPC (exception PC)
    signal status:  std_logic_vector(31 downto 0);  -- Processor status
    signal cause:   std_logic_vector(31 downto 0);  -- Exception cause
    signal badaddr: std_logic_vector(31 downto 0);  -- Bad Addr (mem access)
    signal count:   std_logic_vector(31 downto 0);  -- Count (timer)
    signal compare: std_logic_vector(31 downto 0);  -- Compare (timer)
    signal newip,ip:std_logic_vector(7 downto 0);  -- Interrupt pending
    signal ivecmaskerade: std_logic_vector(7 downto 0);  -- Interrupt flags & masks

    signal causecode: std_logic_vector(4 downto 0);
    signal waitcycles: std_logic_vector(1 downto 0); -- Wait cycles for RFE
    signal clock_interrupt: std_logic;
begin

    -- Interrupt enable
    ie <= status(0) when (waitcycles = "00") else '0';

    -- Clock interrupt
    clock_interrupt <= '1' when (compare = count) else '0';

    -- Interrupts
    newip <= clock_interrupt & extint & "000000";
    ip <= cause(15 downto 8) OR newip;

    ivecmaskerade <= status(15 downto 8) AND cause(15 downto 8);
    int <= '0' when (ivecmaskerade = "00000000") else '1';

    causecode <= "00000" when (int = '1') else ncause;

    -- Register read
    process(rdr,epc,status,cause,badaddr,count,compare)
    begin
        case rdr is
        when CP0_EPC     => dout <= epc;
        when CP0_STATUS  => dout <= status;
        when CP0_CAUSE   => dout <= cause;
        when CP0_BADADDR => dout <= badaddr;
        when CP0_COUNT   => dout <= count;
        when CP0_COMPARE => dout <= compare;
        when others => dout <= (others => '-');
        end case;
    end process;

    process (clk)
    begin
        if (clk'event and clk='1') then
            if (reset = '1') then
                epc <= (others => '0');
                cause <= (others => '0');
                status <= (others => '0');
                count <= (others => '0');
                waitcycles <= (others => '0');
            else
                if (waitcycles /= "00") then
                    waitcycles <= std_logic_vector(unsigned(waitcycles)+1);
                end if;

                count <= std_logic_vector(unsigned(count)+1);

                if (isex = '1' and ie = '1') then
                    epc <= pc;
                    cause <= brslot & "000000000000000" & ip & "0" & causecode & "00";
                    status <= X"0000"&status(15 downto 8)&"00"&status(3 downto 0)&"00";
                else
                    if (rfe = '1') then
```

```
                    status <= X"0000" & status(15 downto 8) & "00" &
                                        status( 5 downto 4) & status(5 downto 2);
                    waitcycles <= "01"; -- Contador de waitcyles
                end if;  -- rfe
                cause <= cause(31 downto 16) & ip & cause(7 downto 0);
            end if;  -- is exception

            if (wen = '1') then
                case wrr is
                when CP0_EPC     => epc <= din;
                when CP0_CAUSE   => cause <= din;
                when CP0_BADADDR => badaddr <= din;
                when CP0_COMPARE => compare <= din;
                when CP0_STATUS  => status <= din;
                when CP0_COUNT   => count <= din;
                when others      =>
                end case;
            end if;  -- wen
        end if;  -- reset
    end if;  -- clk event
end process;

end architecture;
```

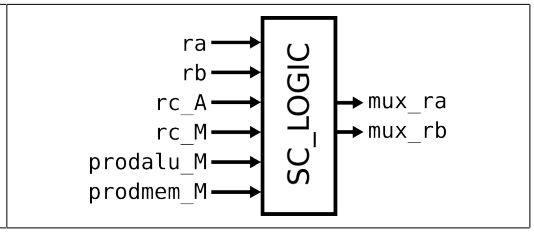
8. Controlador de curtcircuits

El controlador de curtcircuits és el responsable de controlar els multiplexors presents a la sortida del banc de registres a l'etapa *DL*. L'objectiu és seleccionar l'entrada del multiplexor que proporciona el valor correcte del registre al qual s'està accedint. Si no es pot accedir a un valor correcte el comportament no està definit, ja que la unitat de control de riscs s'encarrega de solucionar aquestes situacions.

Aquesta unitat disposa d'informació de diverses etapes. Té accés als identificadors dels registres als quals s'està accedint a l'etapa *DL* (*ra* i *rb*) així com els identificadors de registre destí de les instruccions en les etapes *ALU* i *MEM*: *rc_A* i *rc_M*. Amb aquesta informació la unitat pot determinar si els accessos al banc de registres provoquen un risc i actuar en conseqüència. Donat que les instruccions poden produir les dades en diferents punts del camí de dades les senyals *prodalu_M* i *prodmem_M* indiquen per quin camí viatgen les dades vàlides.

Entitat

```
entity sc_logic is port (
  ra,rb:     in std_logic_vector(4 downto 0);
  rc_A,rc_M: in std_logic_vector(4 downto 0);
  prodalu_M: in std_logic;
  prodmem_M: in std_logic;
  mux_ra:   out std_logic_vector(1 downto 0);
  mux_rb:   out std_logic_vector(1 downto 0));
end entity;
```



Arquitectura

```
architecture arch of sc_logic is
begin
    mux_ra <= "01" when (ra /= "00000" and (ra = rc_A)) else
              "10" when (ra /= "00000" and (ra = rc_M and prodalu_M = '1')) else
              "11" when (ra /= "00000" and (ra = rc_M and prodmem_M = '1')) else
              "00";

    mux_rb <= "01" when (rb /= "00000" and (rb = rc_A)) else
              "10" when (rb /= "00000" and (rb = rc_M and prodalu_M = '1')) else
              "11" when (rb /= "00000" and (rb = rc_M and prodmem_M = '1')) else
              "00";
end arch;
```

9. Control de riscs

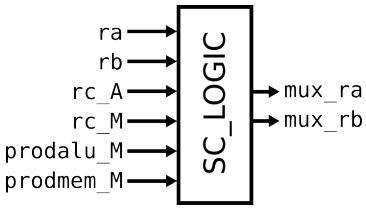
La unitat de control de riscs té la missió de controlar la segmentació i actuar davant de situacions de risc que no es poden resoldre fent ús de curtcircuits. Controla el bloqueig de les quatre primeres etapes del processador així com la injecció de *nop*s en tres etapes més.

La unitat té accés als identificadors de registre que es llegeixen en *DL* (*ra* i *rb*) i els identificadors de registre destí de les etapes *ALU* i *MEM* (*rc_A* i *rc_M*). Disposa addicionalment d'informació sobre els punts de producció de dades de les instruccions que es troben a les etapes *ALU* i *MEM* (*prodalu_A*, *prodalu_M* i *prodmem_M*). Amb aquesta informació i la senyal *jump*, que indica si la instrucció realitza un salt, es pot detectar si es produeix un risc que no es pugui solucionar amb un curtcircuit.

L'actuació es realitza amb les senyals de bloqueig: *stall_cbd* bloqueja les etapes *CP*, *BUS* i *DL* i *stall_alu* bloqueja l'etapa *ALU*. I amb les senyals d'injecció de *nop*s, que a banda de complementar el funcionament dels bloquejos permeten descartar instruccions de la segmentació: *nop_bus_dec* insereix una *nop* a l'etapa *DL*, *nop_dec_alu* injecta una *nop* a l'etapa *ALU* i *nop_alu_mem* n'injecta una a *MEM*.

Entitat

```
entity risk_unit is port (
  ra,rb:       in std_logic_vector(4 downto 0);
  rc_A,rc_M:   in std_logic_vector(4 downto 0);
  isjump:      in std_logic;
  prodalu_A:   in std_logic;
  prodalu_M:   in std_logic;
  prodmem_M:   in std_logic;
  stall_cbd:  out std_logic;
  stall_alu:  out std_logic;
  nop_bus_dec:out std_logic;
  nop_dec_alu:out std_logic;
  nop_alu_mem:out std_logic);
end risk_unit;
```



Arquitectura

```
architecture arch of risk_unit is
    signal riskA, riskB, riskS, riskD : boolean;
begin

    riskS <= isjump = '1';

    riskA <= (ra /= "00000" and (
                    (ra = rc_A and prodalu_A = '0') or
                    (ra = rc_M and ra /= rc_A and prodmem_M = '0' and prodalu_M = '0')
                              )
             );
```

```
    riskB <= (rb /= "00000" and (
                      (rb = rc_A and prodalu_A = '0') or
                      (rb = rc_M and rb /= rc_A and prodmem_M = '0' and prodalu_M = '0')
                                  )
             );

    riskD <= (riskA or riskB);

    process(riskD,riskS)
    begin
        if (riskD and riskS) then
            stall_cbd <= '1';
            stall_alu <= '1';
            nop_bus_dec <= '0';
            nop_dec_alu <= '0';
            nop_alu_mem <= '1';
        elsif (riskD) then
            stall_cbd <= '1';
            stall_alu <= '0';
            nop_bus_dec <= '0';
            nop_dec_alu <= '1';
            nop_alu_mem <= '0';
        elsif (riskS) then
            stall_cbd <= '0';
            stall_alu <= '0';
            nop_bus_dec <= '1';
            nop_dec_alu <= '0';
            nop_alu_mem <= '0';
        else
            stall_cbd <= '0';
            stall_alu <= '0';
            nop_bus_dec <= '0';
            nop_dec_alu <= '0';
            nop_alu_mem <= '0';
        end if;
    end process;
end arch;
```

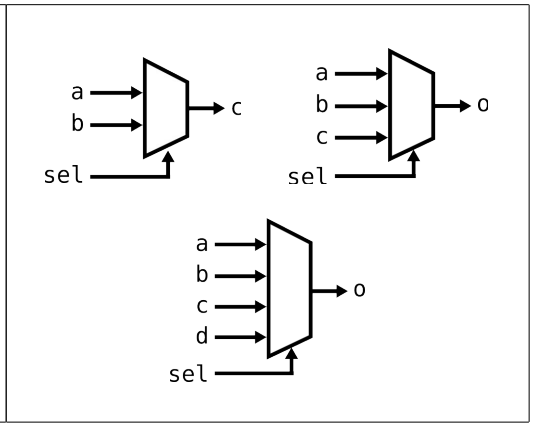
I. Multiplexor

Entitat que, donades diverses dades a la seva entrada, permet el pas d'una d'elles cap a la seva sortida. Es fa servir en múltiples punts del processador per tal de triar entre les dades produïdes per diverses entitats.

S'han implementat multiplexors de busos de 32 bits amb 2, 3 i 4 entrades. Les senyals de control són d'un bit en el primer cas i de dos bits en la resta de casos.

Entitat

```
entity mux32_2 is port (
  a,b: in std_logic_vector(31 downto 0);
  sel: in std_logic;
  c:  out std_logic_vector(31 downto 0));
end entity;

entity mux32_3 is port (
  a,b,c: in  std_logic_vector(31 downto 0);
  sel:   in  std_logic_vector( 1 downto 0);
  o:     out std_logic_vector(31 downto 0) );
end entity;

entity mux32_4 is port (
  a,b,c,d: in  std_logic_vector(31 downto 0);
  sel:     in  std_logic_vector( 1 downto 0);
  o:       out std_logic_vector(31 downto 0));
end entity;
```



Arquitectura

```
architecture arch of mux32_2 is
begin
    c <= a when (sel = '0') else b;
end architecture;


architecture arch of mux32_3 is
begin
    o <= a when (sel = "00") else
         b when (sel = "01") else
         c;
end architecture;


architecture arch of mux32_4 is
begin
    o <= a when (sel = "00") else
         b when (sel = "01") else
         c when (sel = "10") else
         d;
end architecture;
```

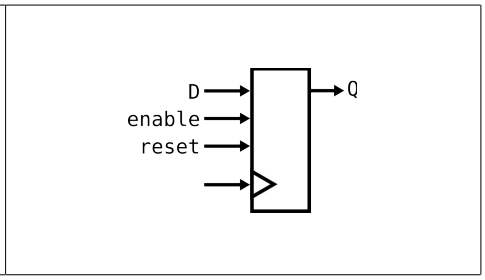
II. Registre

Unitat que permet emmagatzemar un valor binari amb un cert nombre de bits. L'entitat està parametritzada en el nombre de bits, el que significa que es poden crear instàncies amb capacitats diferents.

El registre disposa d'un bus de lectura (*Q*) i d'un altre bus d'escriptura (*D*) així com de senyals de permís d'escriptura (*enable*) i *reset* síncron (*reset*). L'escriptura es realitza per flanc ascendent de la senyal de rellotge (*clk*).

Entitat

```
entity g_register is
  generic ( width : integer := 32 );
  port (
    D:   in std_logic_vector((width-1) downto 0);
    clk:     in std_logic;
    enable:  in std_logic;
    reset:   in std_logic;
    Q:  out std_logic_vector((width-1) downto 0)
  );
end entity;
```



Arquitectura

```
architecture arch of g_register is
begin
    process(clk)
    begin
        if (clk'event and clk='1') then
            if (reset = '1') then
                Q <= (others => '0');
            else
                if (enable = '1') then
                    Q <= D;
                end if;
            end if;
        end if;
    end process;
end arch;
```

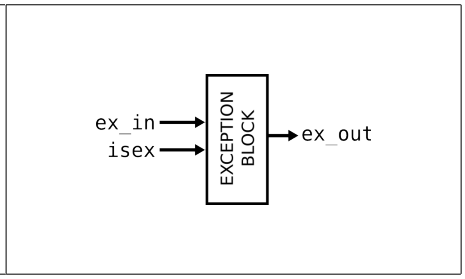
III. Propagador d'excepcions

Aquesta entitat es fa servir al camí de dades del processador per a propagar les excepcions que produeix una determinada instrucció. Està basat en un multiplexor i lògica que controla la seva senyal de selecció.

L'entitat està parametritzada amb un literal de 5 bits que representa el codi d'excepció. Quan es produeix una excepció (*isex* = 1) la sortida (*ex_out*) pren el valor del paràmetre de 5 bits indicat. Això es produeix sempre que l'excepció d'entrada (*ex_in*) sigui diferent de zero, ja que en aquest cas es propaga aquesta.

Entitat

```
entity exception_block is
  generic (
   ex_code: std_logic_vector(4 downto 0) := "00000"
  );
  port (
    ex_in:   in std_logic_vector(4 downto 0);
    isex:    in std_logic;
    ex_out: out std_logic_vector(4 downto 0)
  );
end entity;
```



Arquitectura

```
architecture arch of exception_block is
begin
    process (ex_in,isex)
    begin
        if (unsigned(ex_in) /= 0) then
            ex_out <= ex_in;
        elsif (isex = '1') then
            ex_out <= ex_code;
        else
            ex_out <= (others => '0');
        end if;
    end process;
end architecture;
```

Annex C

Especificació de la placa DE2-115

L'especificació de la placa DE2-115 es pot consultar lliurement a la web del fabricant (*de2-115.terasic.com*). Com a referència s'ha recollit en aquest annex totes les dades rellevants d'aquesta especificació per a facilitat la comprensió del projecte sense haver de recórrer a l'especificació completa de la placa.

1. Senyals utilitzades

Nom	Sentit	Descripció
Oscil·lador		
CLOCK_50	Entrada	Senyal de rellotge de 50MHz
Botons		
KEY[3]	Entrada	Senyal provinent del polsador número 3
Interruptors		
SW[0..15]	Entrada	Senyals provinents dels interruptors número 0 a 15
Connector PS/2 Teclat		
PS2_CLK	Entrada/Sortida	Senyal de rellotge del bus PS/2
PS2_DAT	Entrada/Sortida	Senyal de dades del bus PS/2
Interfície VGA		
VGA_CLK	Sortida	Senyal de rellotge del conversor digital-analògic
VGA_VS	Sortida	Senyal de sincronisme vertical del bus VGA
VGA_HS	Sortida	Senyal de sincronisme horitzontal del bus VGA
VGA_SYNC_N	Sortida	Senyal de sincronisme de l'interval de *blanking* (no s'utilitza)
VGA_BLANK_N	Sortida	Senyal que activa o desactiva l'interval de *blanking*
VGA_R[7..0]	Sortida	Senyals que codifiquen el nivell de color vermell del píxel
VGA_G[7..0]	Sortida	Senyals que codifiquen el nivell de color verd del píxel
VGA_B[7..0]	Sortida	Senyals que codifiquen el nivell de color blau del píxel
Display 7 segments		
HEX0[6..0]	Sortida	Senyals de sortida dels 8 *displays* de 7 segments.
HEX1[6..0]		
HEX2[6..0]		
HEX3[6..0]		
HEX4[6..0]		
HEX5[6..0]		
HEX6[6..0]		
HEX7[6..0]		

2. Interfícies i connectors

Els connectors externs de la FPGA fan ús de diversos connectors i/o interfícies per a accedir a dispositius o busos de dades. A continuació detallem les interfícies i el seu funcionament.

2.1 *Display* de 7 segments

Cada un dels 8 grups de 7 bits està connectat al *display* passant per una resistència. El node comú dels 7 *LED*s està connectat a la tensió de referència, de manera que per encendre un *LED* cal aplicar un nivell lògic baix i per apagar-lo cal aplicar nivell lògic alt. A la *figura C.1* es mostra un esquema.

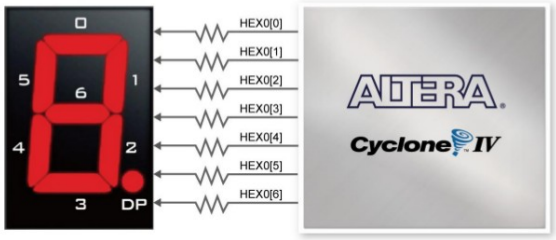


Figura C.1: Esquema de la connexió del display 7 segments

2.2 Interfície VGA

El bus VGA consisteix en tres senyals analògiques corresponents als 3 colors bàsics del píxel que es transmet i dos senyals de sincronisme. Les senyals de sincronisme es transmeten fent ús de nivells *TTL*, pel que la sortida de la *FPGA* està connectada directament al bus VGA. En canvi les senyals analògiques prenen valors entre 0 i 0.7V, pel que és necessari l'ús d'un conversor analògic-digital. Aquest conversor és el *ADV7123* .

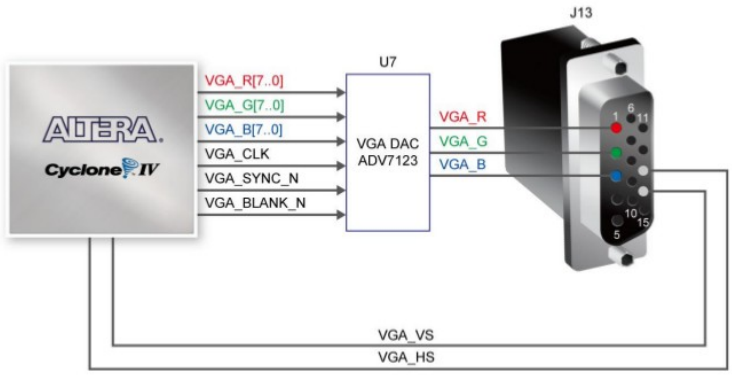


Figura C.2: Esquema de connexió de la FGPA amb el connector VGA

Com es pot veure a la *figura C.2* les senyals digitals que codifiquen el color del píxel es converteixen en valors analògics. La senyal auxiliar de *blanking* permet indicar el *blanking* de la pantalla o de la línia.

2.3 Bus PS/2

El bus PS/2 es un bus molt simple consistent en una senyal de rellotge i una de dades. Fa ús de nivell de tensió TTL però, donat que és bidireccional, es col·loquen resistències al bus per a permetre la senyalització per part dels dos extrems (*figura C.3*)

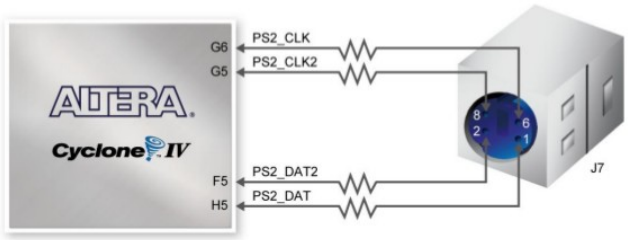


Figura C.3: Esquema del bus *PS/2* connectat a la *FPGA*

Annex D

Anàlisi temporal

En aquest annex es recullen els retards de les diferents entitats del processador. En el cas d'algunes entitats s'ha diferenciat entre els retards a diferents sortides per tal de poder realitzar un càlcul global de retards molt més acurat.

Entitat	Sortides	T1 (ns)	T2 (ns)
next_pc	Totes	3,54	3,17
instruction_mem	Totes	4,91	4,54
maindecoder	Excepcions (1)	1,64	1,94
	Fi d'excepció (2)	2,12	1,51
	Identificadors de registres	3,61	3,34
	Senyalització de producció (3)	1,14	1,02
	Senyals no crítiques (4)	4,13	3,80
regbank	Totes	4,03	3,66
alu	Totes	2,00	1,76
brcontrol	Canvi de PC (*targetjump* i *changepc*)	5,67	5,14
	Adreça del salt (*rettarget*)	2,69	2,37
storefmt	Totes	1,46	1,34
loadfmt	Totes	2,49	2,23
main_mem	Totes	4,91	4,51
memory_decoder	Senyals de l'etapa ALU (5)	1,61	1,45
	Senyals de l'etapa MEM (6)	1,43	1,28
cp0regbank	Totes	3,68	3,42
risk_unit	Totes	1,77	2,53
sc_logic	Totes	2,60	2,37
mux32_2	Totes	0,88	0,79
mux32_3	Totes	1,78	1,61
mux32_4	Totes	1,99	1,77

Notes:

1. Conjunt de senyals que indiquen l'existència d'una excepció: *ukwop*, *syscall*, *brk* i *overflow*.
2. Senyal associada a la instrucció *rfe*.
3. Senyals *prodmem* i *prodalu*, utilitzades per la lògica de riscos i curtcircuits.

4. Conjunt de senyals que es fan servir a les etapes posteriors i que, per tant, es propaguen a registres.
5. Conjunt de senyals que es calculen al final de l'etapa ALU: *uram_wen*, *kram_wen*, *dev_wen*, *ram_ben* i *needfmt*.
6. Conjunt de senyals que es calculen al començament de l'etapa MEM: *sel*.

Annex E

Eines de construcció d'executables

En aquest annex s'introduiran les eines i programes utilitzats per a la construcció d'executables per al computador.

1. Compilador i assemblador

El compilador utilitzat és el *GNU C Compiler* (*GCC*), de codi obert disponible a l'adreça *http://gcc.gnu.org/*. Aquest programa compila codi escrit en el llenguatge C en programes per l'arquitectura MIPS. Per a assemblar programes escrits en assemblador es fa ús del *GNU Assembler*, programa que forma part de *GCC*.

Per a obtenir un compilador per a l'arquitectura MIPS cal descarregar el codi font del compilador i configurar-ne la seva construcció per a que produeixi un compilador *MIPS*. Un cop construït aquest ja es poden compilar programes escrits en C o assemblador i produir executables amb instruccions que el processador és capaç d'executar.

Les ordres de compilació en el cas que ens ocupa serien:

```
mips-elf-gcc -EL -O2 -G0 -nostdlib -mips1 -o fitxer.o -c fitxer.c
```

```
mips-elf-as -EL -o fitxer.o -c fitxer.S
```

Aquestes ordres produeixen fitxers a partir de programes C i assemblador respectivament. S'indica al compilador i l'assemblador que el processador és *Little Endian* amb el paràmetre *-EL*. En el cas de compilar un programa C cal, a més, especificar que el processador té un joc d'instruccions (*ISA*) del *MIPS I* amb el paràmetre *-mips1*. Els paràmetres *-G0 -nostdlib* prevenen al compilador d'utilitzar funcionalitats no disponibles al computador com l'accés relatiu al registre *GP* i l'ús de les llibreries de sistema estàndards. L'opció *-O2* indica la utilització d'un nivell d'optimització del codi produït.

2. Enllaçador (*linker*)

Els fitxers produïts pel compilador i l'assemblador contenen blocs d'instruccions que codifiquen funcions i programes. Però per a produir un programa usable pel processador cal enllaçar aquests fitxers en un sol executable. Per a realitzar aquesta tasca es fa servir el *GNU linker*. Aquest programa permet muntar les instruccions produïdes pel compilador i l'assemblador en un executable tipus ELF (*Executable and Linkable Format*, format d'executable propi dels sistemes Linux i similars).

Per a enllaçar el programa cal tenir en compte l'arquitectura de memòria que s'ha implementat. Els fitxers produïts pel compilador contenen codi i dades que no estan lligades a cap adreça en particular, és a dir, encara no es sap a quina adreça es col·locarà cada instrucció. L'objectiu de l'enllaçador és doncs determinar les adreces de cada instrucció i modificar les instruccions que depenen d'alguna adreça per tal que codifiquin l'adreça adequada a l'accés. Per a realitzar aquesta tasca es fa ús dels anomenats *linker scripts*, que són fitxers amb instruccions que indiquen a l'enllaçador com col·locar les dades i les instruccions a memòria.

A la *figura E.1* es llista el script es fa servir per a enllaçar el programa del nucli.

```
SECTIONS {
    . = 0x80000000;
    .reset_vector : { * (.reset_vector); }
    . = 0x80000008;
    .exception_vector : { * (.exception_vector); }
    . = 0x80000010;
    .text : { * (.text); }
    . = 0xBFFF0000;
    .rodata : { * (.rodata); }
    .data : { * (.data); }
    .bss : { * (.bss); }
}
```

Figura E.1: Script utilitzat per enllaçar el nucli

Com es pot veure el script indica a quina adreça col·locar cada tros del programa. En el cas del nucli es col·loca a les adreces 0x80000000 i 0x80000008 els punts d'entrada de *reset* i excepció respectivament, ja que el processador salta a aquestes adreces de forma fixa. A partir d'aquest punt es col·loquen les instruccions que formen el programa del nucli. Finalment, a partir de l'adreça 0xBFFF0000, es col·loquen totes les variables de memòria, ja que aquest és l'espai de memòria de dades reservat al nucli i el qual és accessible mitjançant operacions de *load*/*store*.

3. Generació de programes

Fent ús de compilador, assemblador i enllaçador s'obté un programa per a l'arquitectura *MIPS* el qual s'adapta a l'organització de memòria del computador. El fitxer obtingut es troba en format *ELF*, format habitualment utilitzat per sistemes operatius compatibles amb Linux. Aquest fitxer conté informació de com carregar el programa que serà utilitzada pel sistema operatiu per a executar-lo.

Per a carregar un programa a la *FPGA* el procés és bastant diferent. Les memòries que es fan servir al computador es troben al disseny d'aquest i s'inicialitzen fent ús de fitxers en un format especificat pel software *Quartus*. Aquests fitxers es fan servir durant la síntesi del disseny i s'inclouen a l'interior del fitxer de programació de la *FPGA*.

És necessari obtenir un fitxer *MIF* (format que usa *Quartus*) per cada memòria a partir de l'executable *ELF* i finalment compilar el disseny i enviar-lo a la FPGA. Per a crear els fitxers *MIF* es fan servir les eines de tractament d'executables anomenades *binutils*. Aquestes eines formen part de l'entorn de desenvolupament de GNU i es poden trobar lliurement a *http://gnu.org/software/binutils/*. Cal descarregar-ne el codi i compilar els programes indicant el tipus de processador que es vol fer servir. Un cop fet això es poden fer servir aquestes eines per tractar programes pel processador *MIPS*.

Utilitzant l'eina *objdump* es pot obtenir el codi i les dades d'un executable en format *ASCII* (codificat en hexadecimal). Això resulta útil ja que el format *MIF* fa servir un format molt similar. Només cal afegir adreces de memòria i capçaleres per a compatibilitzar-ho. Donat que la conversió és senzilla s'ha implementat aquest fent ús de *shellscripts*.

El següent script (*figura E.2*) genera un arxiu *MIF* a partir d'un fitxer *ELF*. Més concretament genera el fitxer *MIF* associat al nucli, com es pot veure per l'ús de seccions associades a aquest (*exception_vector* i *reset_vector*).

```
$(OBJDUMP) -D --section=.text –section=.reset_vector
    --section=.exception_vector $(TARGET) > $(TARGET).txt

echo -e "DEPTH = 16384;\nWIDTH = 32;\nADDRESS_RADIX = DEC;\nDATA_RADIX =
    HEX;\nCONTENT\nBEGIN\n" > $(TARGET).mif

cat $(TARGET).txt | cut -f 1,2  | grep '^[0-9a-f :[:blank:]]*$$' |
    sed '/^$$/d' | awk '{ print FNR-1": " $$2";"}' >> $(TARGET).mif

    echo -e "\nEND\n" >> $(TARGET).mif
```

Figura E.2: Primeres línies de la sortida de *objdump* pel programa “kernel”

A la *figura E.3* es mostra un exemple de sortida proporcionada per *objdump* i a la *figura E.4* es mostra el fitxer *MIF* associat a aquesta. Com es pot veure el fitxer conté les instruccions *MIPS* codificades i les adreces d'aquestes.

```
kernel:     file format elf32-littlemips

Disassembly of section .reset_vector:

80000000 <.reset_vector>:
80000000:      0800017f       j      800005fc <reset_handler>
80000004:      00000000       nop

Disassembly of section .exception_vector:

80000008 <.exception_vector>:
80000008:      08000136       j      800004d8 <exception_handler>
8000000c:      00000000       nop

Disassembly of section .text:

80000010 <kernel_init>:
80000010:      27bdffe0       addiu  sp,sp,-32
80000014:      afb10018       sw     s1,24(sp)
80000018:      3c11bfff       lui    s1,0xbfff
8000001c:      afb00014       sw     s0,20(sp)
80000020:      afbf001c       sw     ra,28(sp)
80000024:      0c0002f0       jal    80000bc0 <init_devices>
80000028:      2630000c       addiu  s0,s1,12
8000002c:      0c0000be       jal    800002f8 <tasks_init>
80000030:      00000000       nop
80000034:      00002821       move   a1,zero
80000038:      24060090       li     a2,144
[...]
```

Figura E.3: Primeres línies de la sortida de *objdump* pel programa “kernel”

```
DEPTH = 16384;
WIDTH = 32;
ADDRESS_RADIX = DEC;
DATA_RADIX = HEX;
CONTENT
BEGIN

0: 0800017f;
1: 00000000;
2: 08000136;
3: 00000000;
4: 27bdffe0;
5: afb10018;
6: 3c11bfff;
7: afb00014;
8: afbf001c;
9: 0c0002f0;
10: 2630000c;
11: 0c0000be;
12: 00000000;
13: 00002821;
14: 24060090;
[...]
```

Figura E.4: Primeres línies del programa “kernel” convertides a format *MIF*

Referències

[1] A. Anand Kumar
Fundamentals of digital circuits Capítol 8

[2] Bob Zeidman
Designing With FPGAs and CPLDs Capítol 2

[3] R.C. Cofer, Ben Harding
Rapid System Prototyping With FPGAs Capítol 14

[4] James Ball
Processor Design: System-On-Chip Computing for ASIC and FPGA Capítol 11

[5] Gaganpreet Kaur
VHDL Basics to Programming Capítol 2

[6] Stuart Sutherland
The Verilog PLI Handbook Capítol 1

[7] William Kafig
VHDL 101: Everything You Need to Know to Get Started Capítol 1

[8] Ángel Grediaga Olivo, José Pérez Martínez
Diseño de procesadores con VHDL Capítol 2

[9] MIPS Technologies Inc.
MIPS32® Architecture for Programmers Volume I: Introduction to the MIPS32® Architecture Capítol 3

[10] John L. Hennessy, David A. Patterson
Computer Architecture: A quantitative approach Capítol 2

[11] MIPS Technologies Inc.
MIPS32® Architecture for Programmers Volume III: The MIPS32® and microMIPS32™ Privileged Resource Architecture

[12] MIPS Technologies Inc.
MIPS32® Architecture for Programmers Volume II: The MIPS32® Instruction Set

[13] Harvey G. Cragon
Computer Architecture and Implementation Capítol 1

[14] Altera Corporation
Quartus II Handbook Version 12.0 Volume 1 Capítols 7 i 11

[15] IEEE Computer Society
1003.1-2008 - IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R))

[16] George Pirocanac , Silicon Graphics Inc .
MIPSproTM N32 ABI Handbook