Implementació de la segmentació d'un processador SISA-3 en una FPGA

Martí Anglada Sánchez

Juny 2013

Índex de continguts

Índex de figures

1. Introducció

1.1 Objectius

Aquest projecte proposa aprofitar el processador senzill realitzat durant l'assignatura PEC i combinar-lo amb els conceptes sobre segmentació lineal de l'assignatura AC2 per a aconseguir una implementació segmentada d'un processador capaç d'executar la majoria del joc d'instruccions SISA-3.

L'objectiu clar és el d'aplicar tots els coneixements adquirits durant les dues assignatures: no només portar a la pràctica els dissenys en una arquitectura concreta, sinó també millorar l'habilitat de programar en llenguatges HDL i d'utilitzar simuladors i analitzadors lògics.

Primerament, es vol fer més realista el sistema d'entrada/sortida del disseny del processador del qual es parteix, implementant una gestió d'interrupcions i de mode sistema/usuari. Durant el transcurs de l'assignatura PEC, això no és possible d'implementar per falta de temps, però és el següent pas segons l'estratègia de disseny que es planteja.

Seguidament, es proposa la implementació en VHDL de la segmentació, la gestió de tots els riscos que aquesta arquitectura comporta a l'hora d'executar un programa i l'ús de les tècniques dels curtcircuits i dels predictors de salts per a mitigar-los el màxim possible. D'altra banda, es vol que aquest disseny sigui funcional en una FPGA, en concret a la placa de desenvolupament DE1 d'Altera.

Per últim, com que el computador SISA no és el que s'estudia a l'assignatura AC2, cal adaptar les idees més genèriques a casos concrets. El projecte fa servir aquest fet per a ampliar les possibilitats de dissenyar lògica nova, i traslladar coneixements d'una arquitectura a una altra.

1.2 Llenguatges de descripció de hardware

Des dels anys 70, la complexitat dels circuits ha augmentat en molt gran mesura, pel que és necessària alguna manera de descriure lògica digital en alt nivell, abstreta de la tecnologia usada en els transistors. D'aquesta necessitat neixen els llenguatges de descripció de hardware (HDLs), llenguatges de programació capaços de documentar les interconnexions i comportament d'un circuit.

Aquesta abstracció s'anomena Register-transfer level (RTL), i modela el circuit en termes del flux de senyals entre els registres i les operacions que s'hi apliquen. El resultat de l'RTL es pot traduir a nivell de portes lògiques mitjançant un sintetitzador que, a la seva vegada pot crear un circuit integrat gràcies a eines de col·locació i encaminament.

Els HDL van ser concebuts per a simular circuits, ja que permetien simular l'execució d'un hardware sense la necessitat de fabricar-lo [1]. L'aparició dels sintetitzadors va portar els HDL a ser molt usats en el disseny de sistemes digitals.

Un dels HDL més importants és el que es fa servir en aquest projecte, el VHDL. Amb una sintaxi basada en Ada, en VHDL es declaren mòduls que es connecten entre ells però tenen la implementació interna amagada. Cada mòdul consisteix en una part anomenada Entity (la declaració d'entrades i sortides del component) i una altra anomenada Architecture (la descripció del comportament del mòdul).

Els senyals de cada mòdul tenen un dels quatre modes següents: In (entrada), Out (sortida), Inout (bidireccional) i Buffer (sortides que també poden ser llegides). Els tipus de dades més usats són el std_logic (escalar) i std_logic_vector (cadena d'elements std_logic), en què cada element pot prendre un de 9 possibles valors: '1' fort i '1' dèbil, '0' fort i '0' dèbil, alta impedància, desconegut i desconegut dèbil, valor sense importància o valor no inicialitzat [2].

El llenguatge és fortament tipat i permet que l'usuari defineixi els seus propis tipus de dades. Aquesta rigor amb els tipus dificulta que no es puguin detectar errors de modelatge, però fa que el codi hagi de ser llarg. Permet usar llibreries i paquets, i disposa de sentències per a replicar instàncies de la mateixa unitat i connectar-les correctament.

El VHDL té un domini seqüencial, executant les instruccions en l'ordre en què apareixen al codi, i un domini concurrent, que permet executar alhora instàncies, assignacions, processos o crides.

A la figura següent podem observar, com a exemple senzill, la implementació en VHDL d'un biestable:

```
LIBRARY IEEE;
USE ieee.std_logic_1164.all;

Entity biestable IS
    Port(D, clock : in std_logic;
    Q : out std_logic
    );
End biestable;

ARCHITECTURE behavior OF biestable IS
BEGIN
    PROCESS (D,Clock)
    BEGIN
        IF (rising_edge(clock)) THEN
            Q <= D;
        END IF;
    END PROCESS;
END behavior;
```

Figura 1: Biestable en VHDL

És visible la similitud amb Ada quant a la sintaxi, la distinció entre la declaració del component i el seu comportament, els diferents modes que prenen les seves senyals i l'ús de la paraula clau "Process" per a indicar un conjunt d'instruccions seqüencials.

1.3 FPGA

Les FPGA (de l'anglès Field Programmable Gate Array) són uns dispositius nascuts a mitjans dels anys 80 que combinen les tecnologies ASIC i PLD. Per una banda, els ASIC (Application-Specific Integrated Circuit) ofereixen una alta optimització de recursos per a l'execució d'una aplicació concreta, mentre que els PLD són components que no tenen una funció definida a l'hora de fabricar-se, sinó que es reconfiguren per l'usuari. La majoria d'FPGA poden, a més, ser parcialment programades, combinant microprocessadors, controladors i blocs lògics configurables en el paradigma anomenat Reconfigurable Computing, que permet adaptar el hardware en temps d'execució.

A diferència dels primers dispositius lògics programables com els PLD i els PAL, les FPGA no estan composades per portes AND i OR, sinó que utilitzen blocs lògics (LE, Logic Elements) per a implementar les funcions que siguin necessàries. Al poder expressar una funció com a equació booleana i, aquesta, com una taula de veritat, les FPGA utilitzen taules de cerca (LUT, Lookup Table) per a assignar valors depenent de les entrades.

Els blocs bàsics de les FPGA estan formats per LUTs com a elements combinacionals i un biestable que permet guardar la sortida de la LUT. La sortida del bloc pot ser la sortida del biestable o la sortida de la LUT. A la figura 2 es mostra el bloc bàsic de la Cyclone II, la FPGA usada en aquest projecte:

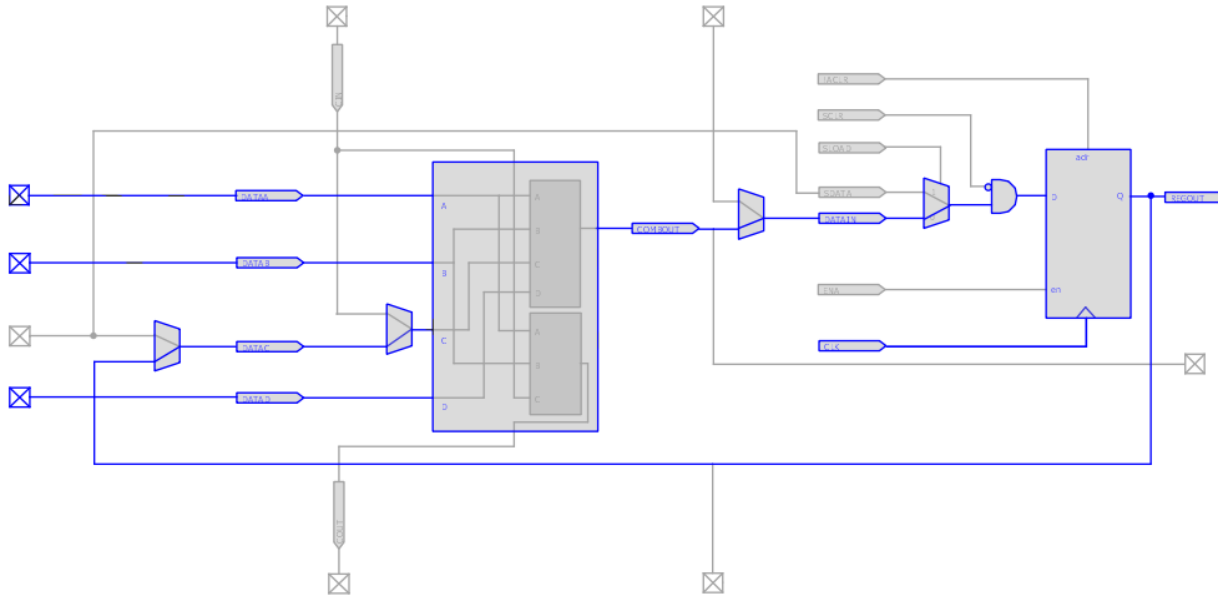


Figura 2: Bloc bàsic de la Cyclone II

Aquest bloc es divideix en part combinacional i part seqüencial. La primera, inclou dues LUT (una de 4 entrades i una altra de 3) i, la segona, permet guardar una entrada o el resultat d'una funció.

Per a connectar els blocs bàsics, s'utilitza matriu de busos d'interconnexió programables, com es veu a la figura següent:

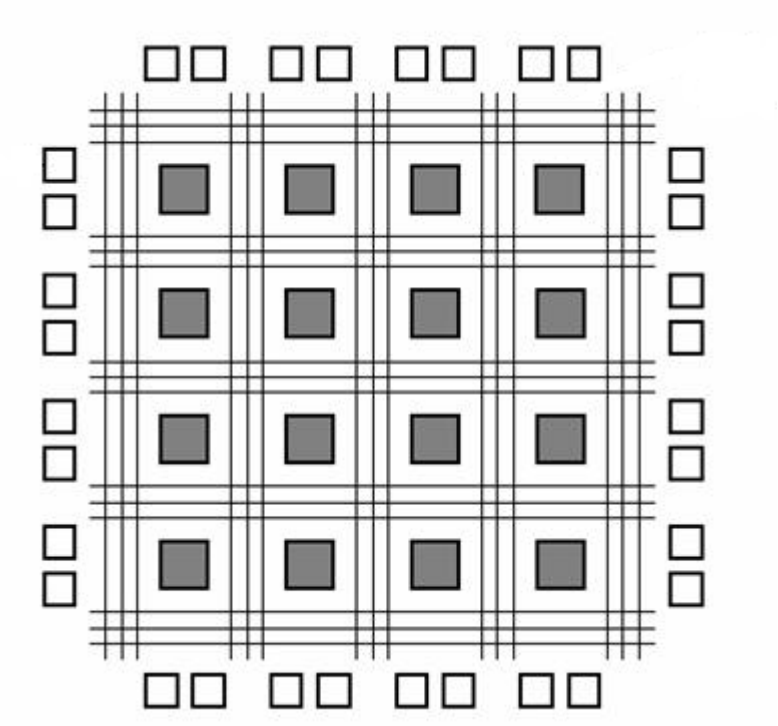


Figura 3: Arquitectura d'una FPGA

A l'interior de la imatge es veuen els blocs bàsics, rodejats de les interconnexions. En un anell exterior es troben els blocs d'entrada/sortida, que permeten comunicar la FPGA amb elements exteriors. En forma de matriu i comunicant els elements, estan els busos d'interconnexió.

Aquesta arquitectura provoca que els dispositius siguin concurrents, pel que ofereixen una gran capacitat de processament. A més, a l'utilitzar una freqüència menor que una CPU, el consum és menor. Per tant, en un entorn no genèric i optimitzat, les FPGA poden aconseguir un rendiment computacional elevat i una bona relació MIPS/Watt.

Les FPGA s'acostumen a fer servir en el camp del processament digital del senyal degut a la seva capacitat de treball en paral·lel i el seu reduït cost en comparació als ASIC. Tot i això, les FPGA també tenen aplicacions en camps tan diferents com la codificació, els sistemes de visió artificial o el tractament d'imatges mèdiques.

1.4 Estat de l'art

A l'iniciar el projecte, no hi havia cap implementació d'un computador segmentat que reconegués el joc d'instruccions SISA-3. Sí que es podien trobar multitud de System on Chip ja implementats (com l'STORM [3]), processadors segmentats (com el mips r2000 [4]) o una implementació del SISA-3 funcional per una FPGA [5]. Però no existia cap implementació segmentada del SISA-3 i, tot i que ja hi havia processadors que incorporaven el que volia fer aquest projecte de manera més eficient fent servir altres arquitectures, l'objectiu principal era segmentar el SISA i aprendre i consolidar coneixements en el procés.

A l'apartat 2.2 es descriurà l'estat del processador al començar al projecte, és a dir, el producte resultant de l'assignatura PEC i a partir del qual neix aquest.

1.5 Entorn de treball

Per a la realització d'aquest projecte, eren necessaris tres programes per a treballar i la placa per a implementar físicament el processador. Aquests programes han fet possible l'acompliment de tres etapes del desenvolupament d'una aplicació:

- Escriptura del codi: Quartus II 11.2. A més de ser un editor potent dissenyat específicament per a escriure codi en HDL, aquest programa inclou el seu propi visualitzador RTL, eina imprescindible en l'escriptura de hardware.
- Proves: ModelSim 10.0c. Simulador molt senzill de fer servir però amb una gran funcionalitat, ideal per a aquest projecte. La eina permet seleccionar quin subconjunt de senyals es vol veure evolucionar i amb quina freqüència evolucionen.
- Depuració: SignalTap II, l'analitzador lògic digital d'Altera. Els simuladors no són perfectes i, per tant, que algun disseny hi funcioni no vol dir necessàriament que ho faci també a la placa final. És imprescindible una eina que permeti capturar els senyals de la placa i mostrar-los en forma de diagrames temporals per a poder detectar possibles problemes.

Aquests programes han estats fets servir en un Core 2 Duo E4400, processador amb dos nuclis treballant a 2 GHz i 2 GB de memòria RAM en un Sistema Operatiu Windows XP.

El dispositiu emprat és la placa de desenvolupament DE1 d'Altera, que inclou una FPGA Cyclone II. El motiu d'aquesta decisió és, per una banda, que era la placa en què s'havia comprovat la funcionalitat del processador del que partia el projecte i, per tant, no causava problemes de compatibilitat a l'iniciar-lo. D'altra banda, l'elecció d'Altera és per motius econòmics: les seves eines de desenvolupament (les mencionades anteriorment que s'usen en el projecte) es poden obtenir de forma gratuïta, mentre que les més complertes i potser intuïtives eines de Xilinx es van descartar per l'elevat preu d'obtenir-ne la llicència. Entre les plaques d'Altera, la DE1 era una de les que tenia el cost menor, però incloïa característiques necessàries per a poder executar el projecte i, a la vegada, comptava amb dispositius com LEDs i un display de set segments útils per a tasques de depuració.

1.6 Estructura del document

Els continguts de la memòria estan organitzats de la forma següent:

Al capítol 2, es presenta el disseny del processador: quin era l'estat al començar el projecte i totes les modificacions que s'hi han volgut afegir. Al capítol 3, s'explica com s'han implementat els dissenys previs, així com els mètodes per a validar el funcionament del processador. Al capítol 4, es troben els resultats del projecte: la comparativa amb el processador de partida. Al capítol 5, apareix l'anàlisi temporal i econòmic del projecte: com s'ha gestionat el temps per a realitzar-lo i quant ha costat. Al capítol 6, s'explica l'impacte ambiental i legal que té el projecte. Finalment, les conclusions del projecte estan al capítol 7. S'adjunta, a més, la bibliografia del projecte al capítol 8 i un parell d'annexos amb informació complementària: la descripció de l'assoliment de les competències tècniques associades i un glossari de les abreviatures usades en aquesta memòria.

2. Descripció del processador

En aquest capítol es començarà explicant el llenguatge màquina que interpreta el processador i es continuarà detallant les funcionalitats dels mòduls que el composaven en el moment en què comença el projecte. A partir d'aquí, s'explicaran els dissenys que composen el projecte: la gestió d'interrupcions, el pipeline i els mecanismes per a reduir la latència efectiva de les instruccions.

2.1 Llenguatge màquina

El llenguatge SISA-3, està format per un conjunt reduït de 55 instruccions de mida fixa (una paraula de 16 bits) [6]. Aquestes instruccions es poden dividir en diferents grups segons els camps que fan servir:

- Format 3 registres: operacions aritmètiques, lògiques i de comparació. S'usen dos registres com a operands font i un tercer com a destí.
- Format 2 registres: instruccions de salts absoluts, d'accés a memòria, de suma d'un registre amb un immediat i instruccions que mouen dades dels registres de sistema als d'usuari.
- Format 1 registre: instruccions de salts relatius al PC, de moviment d'immediats o d'entrada/sortida.

A la següent figura, s'observa la distribució dels 16 bits de la instrucció segons el seu format:

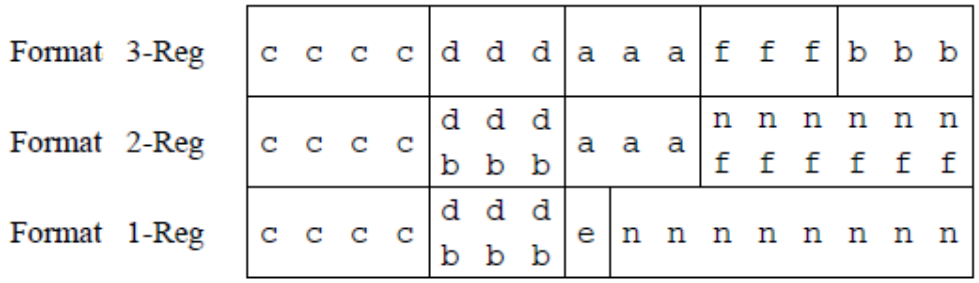


Figura 4: Formats de les instruccions SISA-3

Els 4 bits de més pes de la instrucció (*cccc)* codifiquen el codi d'operació. Els camps *aaa* i *bbb* indiquen registres font, mentre que el camp *ddd* indica un registre destí. El SISA-3 té separats en bancs diferents 8 registres de propòsit general, 8 de coma flotant i 8 de sistema (tots ells de 16 bits), pel que s'utilitzen 3 bits per a adreçar-los. El bit *e* serveix per a codificar possibles extensions del codi d'operació i, els bits *fff*, codifiquen diferents funcionalitats pel mateix codi d'operació. Per últim, els bits *n* indiquen immediats. Les instruccions de sistema que mouen dades entre modes tenen format 2-Reg però el primer dels bits *f* és un bit *e* amb valor '1'.

Les instruccions també es poden dividir segons la seva funcionalitat:

- Instruccions d'operació: Inclou, primerament, les instruccions que realitzen les operacions lògiques bit a bit and, or, not i xor (instruccions AND, OR, NOT i XOR, respectivament). També els desplaçaments lògics i aritmètics, (SHL, SHA) que desplacen un operand tants bits a l'esquerra o a la dreta com indiqui l'altre operand, interpretat com a número en complement a 2 (fent que els números positius desplacin a l'esquerra i, els negatius, a la dreta). A continuació, les instruccions ADD, SUB, MUL i DIV que, respectivament, sumen, resten, multipliquen (obtenint la part baixa) i divideixen (obtenint la part entera) dels dos operands font, interpretats com a números en complement a 2. Per últim, existeixen lleugeres variants d'aquestes operacions: l'ADDI suma un registre font amb un immediat de 6 bits estès de signe, MULH obté la part alta de la multiplicació dels dos operands i, MULHU i DIVU, operen de la mateixa manera que MULH i DIV però interpretant els operands com a naturals (unsigned).
- Instruccions de càrrega d'immediats: Es disposa de les instruccions MOVI i MOVHI. La primera, mou l'immediat de 8 bits i el guarda al registre destí, estenent-lo de signe. La segona, actualitza el registre font, sobreescrivint els 8 bits de més pes del registre.
- Instruccions de comparació: El SISA-3 inclou les instruccions CMPLT, CMPLE i CMPEQ per a respectivament, comparar menor que, menor o igual que i igualtat en enters i les instruccions CMPLTU i CMPLEU per a comparar menor que i menor o igual que en naturals.
- Instruccions d'accés a memòria: Aquestes instruccions fan servir el mode d'adreçament registre base més desplaçament. L'immediat que se suma, al que se li estén el signe, és de 6 bits. La instrucció LDB llegeix un byte de memòria i el carrega en els 8 bits de menys pes del registre destí. La instrucció STB escriu el byte de menys pes del segon registre font a memòria. També es pot llegir i escriure a nivell de word (16 bits) amb les instruccions LD i ST. En aquest cas, però, l'immediat es multiplica per 2 per a evitar adreces no parelles, ja que no s'admeten.
- Instruccions de ruptura de seqüència: Aquest llenguatge màquina té dues maneres de saltar. La primera, relativa al Comptador de Programa, està composada per instruccions que porten un immediat de 8 bits codificat. Aquest serà multiplicat per 2 i sumat al PC per a obtenir l'adreça de salt, que serà la nova adreça del programa si el registre font val zero en el cas de la BZ o si el registre font té un valor diferent de zero en el cas de la BNZ. La segona, utilitza adreces absolutes, és a dir, el valor d'un registre font és copiat al PC. La

còpia pot ser condicional respecte la comparació amb zero del segon registre font en el cas de les instruccions JZ i JZN o incondicional, com les instruccions JMP i JAL. Aquesta última, no només copia el valor d'un registre font al PC, sinó que també es guarda en el registre destí el valor de la adreça corresponent a la següent instrucció en seqüència.

- Instruccions de sistema: S'utilitzen les instruccions EI i DI per a habilitar i inhabilitar interrupcions. Dins del mode sistema, les instruccions RDS i WRS s'utilitzen, respectivament, per a llegir o escriure en un dels 8 registres de sistema. La instrucció GETIID obté l'identificador de la interrupció generada i el guarda el registre destí. S'utilitzen les instruccions WRPI i WRVI per a escriure al tag físic o virtual del TLB d'instruccions i, les WRPD i WRVD, per a fer-ho en el de dades. Per últim la instrucció RETI torna al mode usuari, restaurant el PC i els flags del processador.

2.2 Estat inicial

En començar el projecte, el processador del que es disposa és el resultat de l'assignatura PEC, capaç d'executar el subconjunt d'instruccions del joc SISA-F que no inclou les instruccions de sistema ni operacions en coma flotant. En particular, el processador és funcional a la placa DE1 d'Altera, i fa servir 64 KB dels seus 512 KB de SRAM com a memòria de dades i programa. El processador triga dos cicles a executar cada instrucció, ja que triga un cicle en buscar-la (fetch) i un altre en descodificar-la, executar-la i guardar els resultats a memòria o als registres. L'espai d'adreçament de memòria està separat de l'espai dels ports d'entrada i sortida , que està dividit en dos (INPUT i OUTPUT) amb 2^8 ports de 16 bits cadascun.

Amb el que està implementat, es permet fer entrada/sortida amb els botons, interruptors, LEDs i visors de 7 segments dels que disposa la placa, però és necessari programar per enquesta l'obtenció de dades d'entrada.

El processador fa servir diversos mòduls, com es pot observar a la Figura 6:

- Banc de 8 registres de 16 bits amb dos ports de lectura i un d'escriptura.
- ALU, la unitat que executa les operacions segons els valors de dues senyals d'entrada.
- Mòdul multicicle, encarregat de generar senyals per a llegir una instrucció durant un cicle i executar-la correctament al següent.

- Descodificador, que rep la instrucció buscada a memòria i genera els senyals necessaris, com el permís d'escriptura de memòria i registres o l'operació a realitzar a l'ALU.
- Datapath, mòdul encarregat de fer arribar les entrades correctes al banc de registres, ALU i memòria segons les descodificacions.

A més, són necessaris un controlador de memòria, que genera els senyals necessaris en l'interval de temps requerit per la placa, un controlador pels dispositius d'entrada/sortida, que rep l'adreça del port que es vol llegir o escriure i es comunica físicament amb la placa, així com un mòdul superior a la jerarquia que instanciï els pins de la FPGA.

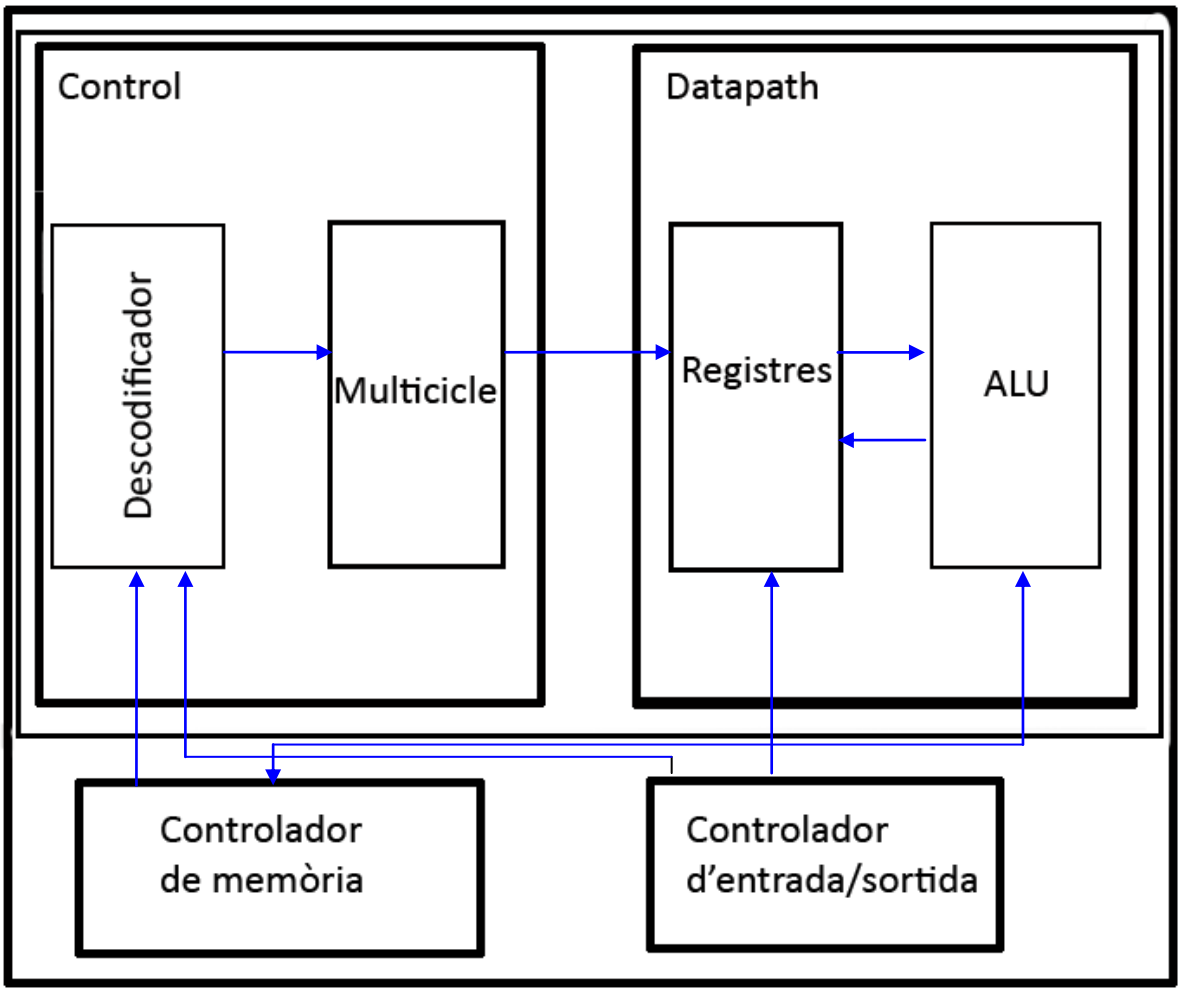


Figura 5: Esquema de mòduls del processador original

2.3 Mode sistema

La primera modificació del projecte és l'addició d'un mode privilegiat (mode sistema). Aquest mode té accés a tota la memòria i hardware, així com la possibilitat d'executar totes les instruccions del llenguatge, mentre que el mode no privilegiat (mode usuari) només en pot fer servir una part. La raó darrere d'aquesta separació és la protecció: com una aplicació d'usuari no pot fer accions que puguin alterar l'estat de la màquina, es garanteix la seguretat del sistema i de les altres aplicacions.

El projecte necessita el mode sistema en l'entorn del tractament d'interrupcions, en particular, els dispositius d'entrada/sortida. Una interrupció és un senyal enviat al processador per a indicar que un esdeveniment necessita la seva atenció. És necessari que el processador pugui rebre les dades dels dispositius mitjançant un mecanisme diferent al d'enquesta, que és la única manera possible al començar el projecte. L'enquesta implica que el processador no realitza cap altra tasca excepte esperar les dades síncronament mentre aquestes no hagin arribat, el que no utilitza eficientment els cicles de CPU.

Per a implementar el mode sistema i les interrupcions, calen les definicions de les estructures on es guarda la informació necessària per a tractar-les i una descripció del pla d'acció en cas de rebre'n una.
Es faran servir 8 registres addicionals (anomenats "S0..S7", en contraposició als anteriors "R0..R7", de propòsit general) de sistema, que no podran ser llegits o escrits en mode no privilegiat. Les funcions d'aquests són:

S0: Conté la paraula d'estat que tenia el sistema quan es va produir la interrupció.

S1: Conté l'adreça de retorn de la interrupció.

S2: Conté un codi que indica el tipus d'esdeveniment que s'ha produït.

S3: Conté, en el cas de fallada de TLB, l'adreça de memòria que l'ha produït.

S4: Variable temporal usada pel sistema.

S5: Conté l'adreça de memòria de sistema on es troba la següent instrucció a executar després de produir-se la interrupció.

S6: Punter a la pila de sistema.

S7: Conté l'estat del processador. En particular, el mode de treball (usuari o sistema) i la inhibició de les interrupcions.

La numeració d'aquests registres segueix els passos que s'han d'executar en cas de rebre una interrupció, és a dir, guardar la paraula d'estat (S0), guardar l'adreça de retorn (S1), escriure el codi de l'esdeveniment (S2), carregar al PC l'adreça d'entrada al sistema (S5) i, per últim passar a mode sistema i inhibir les interrupcions (S7). Els registres S3 i S6 no s'usaran en el projecte perquè no s'implementarà el TLB (i, per tant, no hi haurà fallades de TLB) ni es permetrà l'execució rutines d'interrupció niades.

2.4 Pipeline

El disseny del pipeline del SISA-3 correspon a la figura 6. En els següents apartats es detallarà la funcionalitat de cada etapa, mentre que ara s'explica una visió general de la segmentació.

El processador està segmentat en 6 etapes, que s'executen consecutivament:

1. PC: S'actualitza el comptador de programa.
2. Fetch: S'accedeix a memòria per a llegir la següent instrucció a executar.
3. DR: Es descodifica la instrucció llegida i es llegeixen els operands font.
4. ALU: A les instruccions aritmeticològiques, es fa la operació necessària. A les instruccions de salt condicional, s'avalua la condició. A les instruccions de salt absolutes, es calcula l'adreça efectiva del salt. A les instruccions d'accés a memòria, es calcula l'adreça efectiva.
5. MEM: Les instruccions de memòria accedeixen a memòria per a llegir o escriure dades.
6. WB: S'escriu al banc de registres.

Durant l'etapa DR, es descodifiquen senyals de control que seran necessàries en etapes posteriors. Aquestes es propaguen pel pipeline fins a arribar a l'etapa corresponent, on són utilitzades. Això s'indica al diagrama dividint els registres de desacoblament en diferents blocs: els blocs etiquetats representen l'agrupació de senyals de control específiques que es fan servir en una etapa, mentre que el bloc no etiquetat representa les línies de control generals del processador.

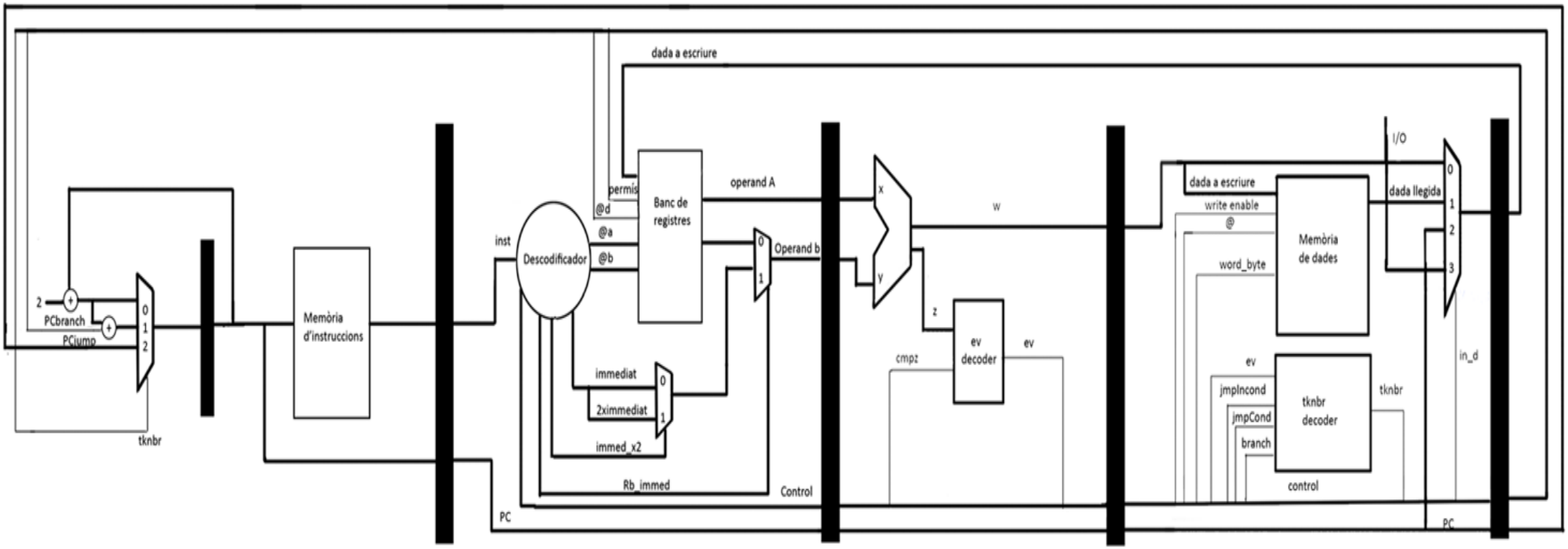


Figura 6: Disseny del pipeline

A la figura 6 també es poden observar els tres bucles hardware del disseny:

1. Fetch-> PC: Ús del valor del comptador de programa a la segona etapa per a calcular el nou valor del comptador de programa a la primera etapa. Aquest càlcul correspon al seqüenciament implícit.
2. WB -> DR: Escriptura al banc de registres.
3. WB -> PC: Escriptura al comptador de programa del valor obtingut durant l'execució d'una instrucció de salt.

2.4.1 Program Counter

S'actualitza el valor del comptador de programa, que pot ser l'anterior + 2 (seqüenciament implícit), el valor del PC més un desplaçament (instruccions branch) o un valor d'entrada que rep l'etapa (instruccions jump). La decisió d'escriure un dels tres valors es pren amb el multiplexor MUX_PC, controlat per un senyal provinent de l'etapa d'escriptura. Aquest senyal té en compte el tipus d'instrucció i, en el cas de ser un salt condicional, si s'ha complert la condició.

La informació que es passa a la següent etapa és només el valor del PC que, a l'iniciar-se el processador, val 0xC000.

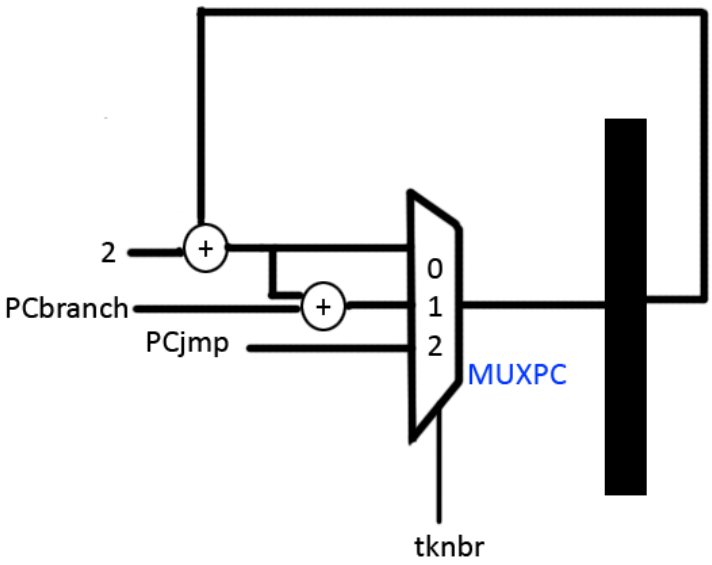


Figura 7: Detall de l'etapa PC

2.4.2 Fetch

S'accedeix a la memòria que emmagatzema el codi mitjançant l'adreça guardada al comptador de programa. La informació que es passa a la següent etapa és la instrucció llegida i el PC.

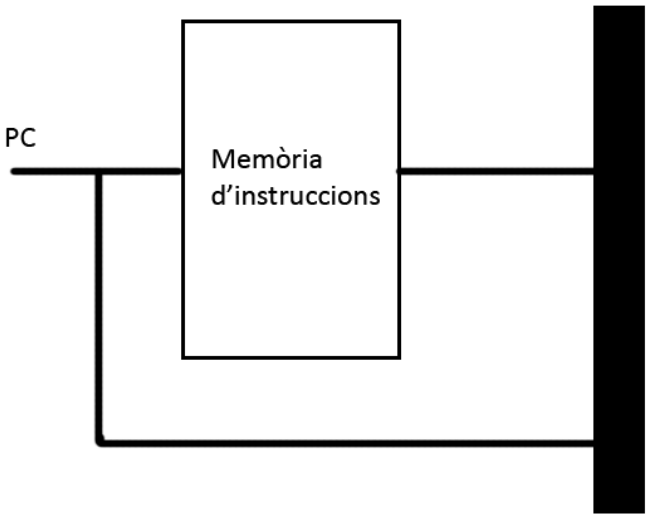


Figura 8: Detall de l'etapa F

2.4.3 Decode & Read

Els components d'aquesta etapa són el descodificador i el banc de registres. El primer rep com a entrada la instrucció llegida a l'etapa anterior, i s'encarrega de generar tot el conjunt de senyals necessaris per a controlar els components usats en les etapes posteriors. Del descodificador també surten dues adreces que s'utilitzen com a entrada del banc de registres per a llegir els operands font.

Al tenir les instruccions mida fixa i els identificadors dels registres font a les mateixes posicions, s'accedeix sempre al banc de registres amb adreces que poden o no ser vàlides. El multiplexor MUXRb_immed s'encarrega de seleccionar el valor correcte del segon operand segons el tipus d'instrucció.

La funció del multiplexor MUX_x2 és la de seleccionar l'immediat correcte que necessita la instrucció. Les instruccions Load word i Store word han d'accedir a posicions parelles però, per a aconseguir adreçar més memòria, a l'immediat en llenguatge màquina se li elimina l'últim bit (que sempre val '0'). Per tant, a l'interpretar la instrucció cal multiplicar aquest immediat per dos (és a

dir, tornar-li a afegir un ‘0’ al final) i cal un control de quin immediat s’ha de fer servir.

La informació que es passa a la següent etapa és: els dos operands que es faran servir a l’ALU, el comptador de programa, els permisos d’escriptura de memòria i registres i tots els senyals de control dels components posteriors.

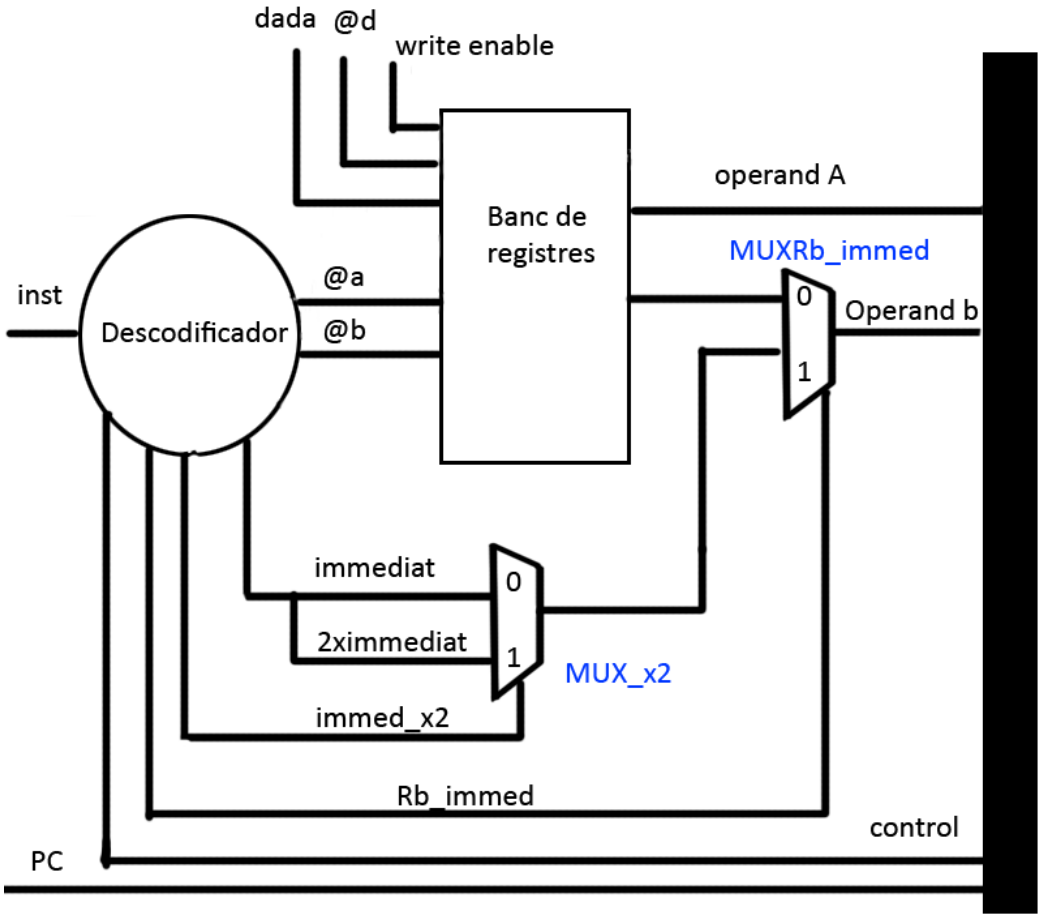


Figura 9: Detall de l'etapa DR

2.4.4 Execution

En aquesta etapa es fan càlculs amb les dades subministrades per l'etapa anterior. Es fa servir l' ALU per a realitzar les operacions aritmeticològiques específiques de l'operació, així com calcular l'adreça destí d'un salt o l'adreça d'accés a memòria de dades. Una de les sortides de l'ALU és un senyal que indica si el resultat de l'operació és '0' o no. Amb aquest senyal i un senyal de control d'aquesta etapa que indica quina comparació fa una instrucció de salt condicional, es calcula el senyal d'avaluació de la condició.

La informació que es passa a la següent etapa és: el comptador de programa, els permisos d'escriptura de memòria i registres i tots els senyals de control dels components posteriors.

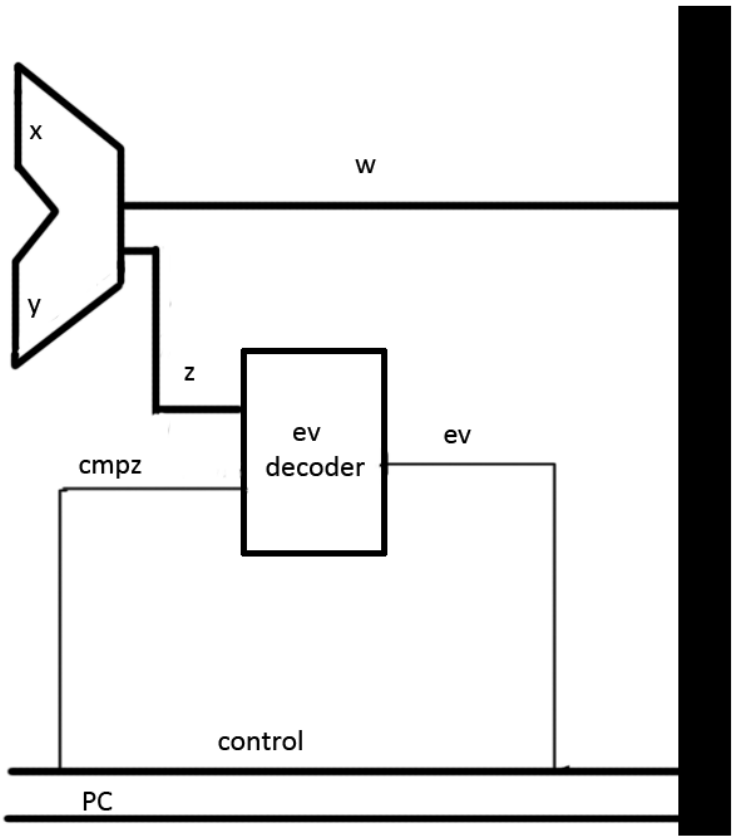


Figura 10: Detall de l'etapa ALU

2.4.5 Memory

Les instruccions Load i Store accedeixen a memòria, usant els senyals de permís d'escriptura, dada a escriure, adreça i granularitat de l'accés (word o byte).

Amb el resultat de l'avaluació de la condició calculat a l'etapa anterior i senyals que indiquen el tipus de salt (condicional o incondicional, Jump o Branch) es calcula el senyal que controla el multiplexor MUX_PC de l'etapa PC.

El multiplexor MUX_d selecciona la dada que s'ha d'escriure al banc de registres, que pot ser el resultat de l'ALU, la dada llegida de memòria, el valor del PC (en la instrucció JAL) o el valor llegit dels dispositius d'entrada/sortida.

La informació que es passa a la següent etapa és: les dades i senyals de control que l'etapa WB passa a l'etapa DR i les dades i senyals de control que l'etapa WB passa a l'etapa PC.

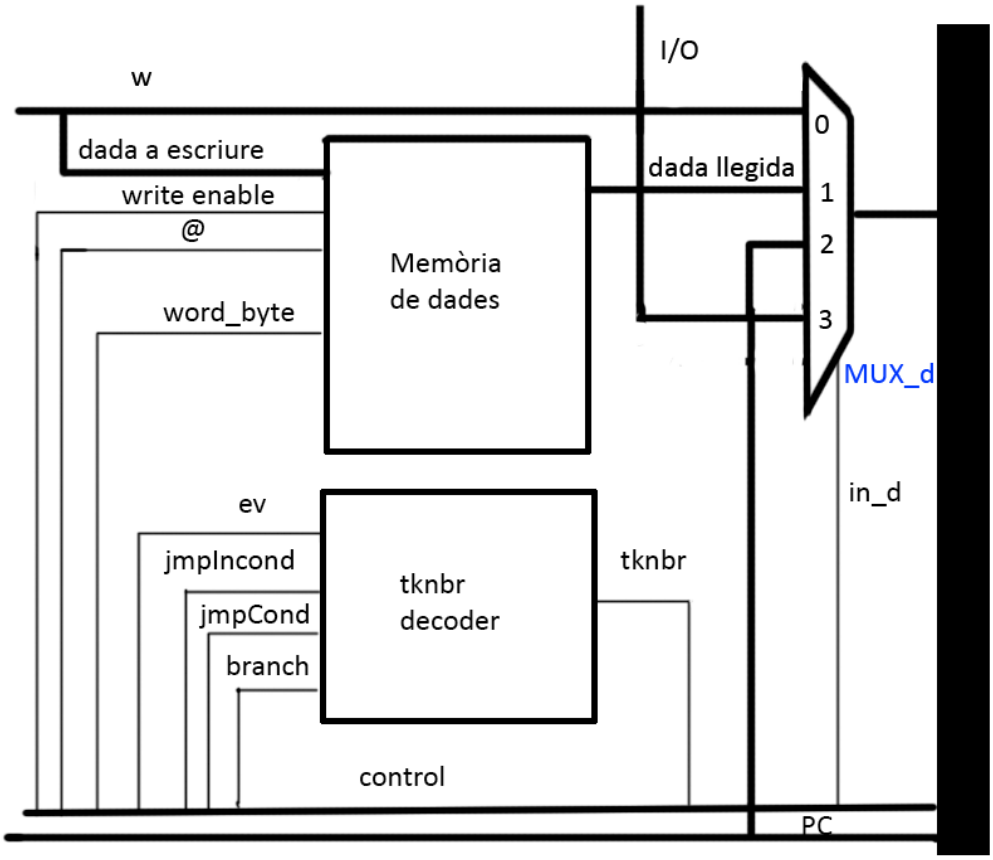


Figura 11: Detall de l'etapa MEM

2.4.6 Write-Back

Aquesta etapa envia a l'etapa DR l'adreça del registre a escriure, la dada a escriure al banc de registres i el seu permís. D'altra banda, a l'etapa PC, passa el senyal de control del multiplexor MUX_PC i l'adreça destí del salt, tant el de les instruccions Jump com el de les instruccions Branch.

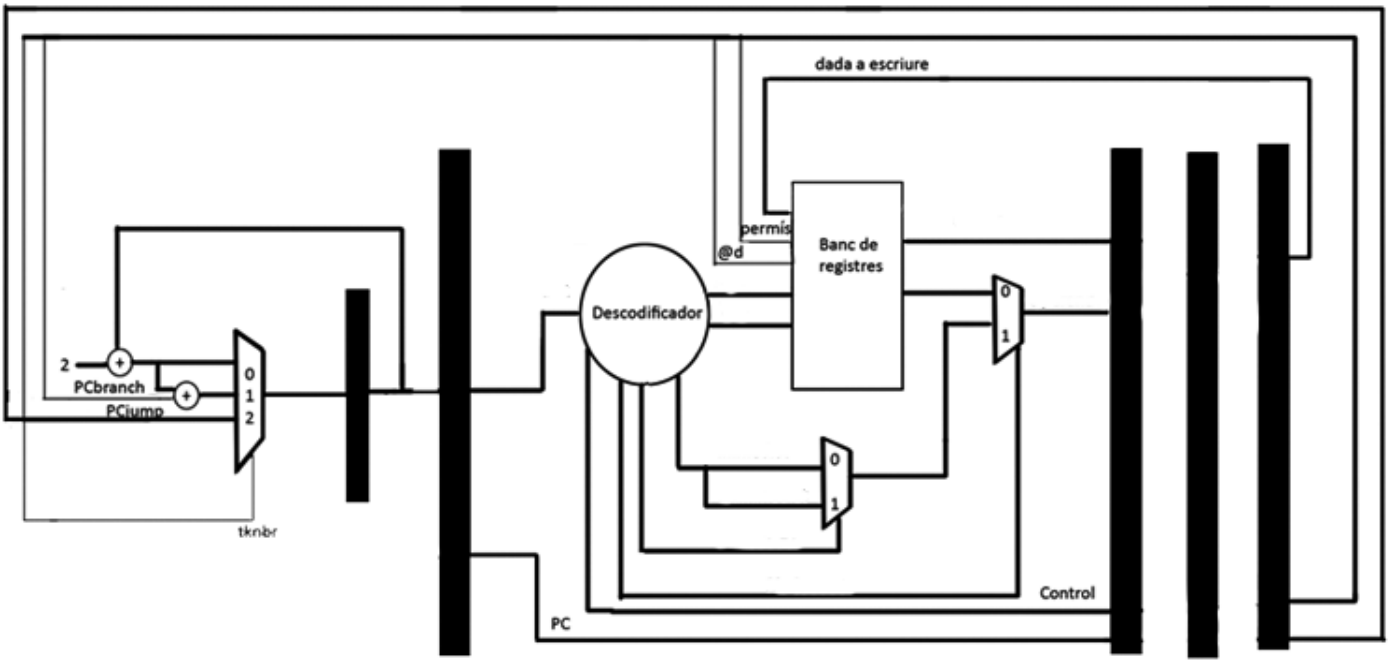


Figura 12: Detall de l'etapa WB

2.5 Gestió dels riscos

2.5.1 Riscos de dades

Un risc de dades és la detecció d'una dependència de dades entre instruccions executades concurrentment on la segmentació no respecta l'ordre font-destí de la dependència. Existeixen 3 tipus de dependències: les dependències vertaderes, les antidependències i les dependències de sortida.

Les antidependències (dependències WAR) tenen lloc quan una instrucció escriu un valor en una posició d'emmagatzematge que una instrucció més vella llegeix. Pot donar-se un problema si la instrucció jove escriu el valor abans que la instrucció vella l'hagi llegit.

```
ADD R0, R1, R2
ADD R1, R2, R2
```

Figura 13: Exemple d'una antidependència

Les dependències de sortida (dependències WAW) tenen lloc quan una instrucció escriu un valor en una posició d'emmagatzematge que una instrucció més vella també escriu. Pot donar-se un problema si la instrucció jove escriu el valor abans que la instrucció vella l'hagi escrit.

```
ADD R1, R0, R2
ADD R1, R2, R2
```

Figura 14: Exemple d'una dependència de sortida

Tal i com està dissenyat el processador, aquests dos tipus de dependències no ocasionen riscos de dades, ja que totes les instruccions triguen el mateix a ser interpretades i passen per les mateixes etapes en el mateix ordre. Per tant, una instrucció vella sempre llegeix els operands abans que una instrucció més jove escrigui al banc de registres o a memòria i una instrucció vella sempre escriu al banc de registres o a memòria abans que una instrucció més jove ho faci.

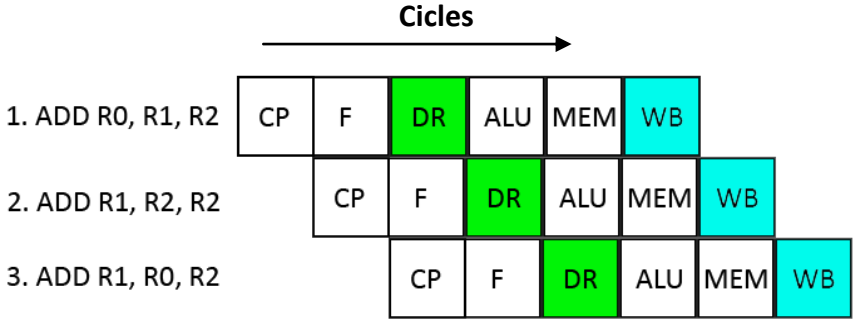


Figura 15: Inexistència de riscos per dependències WAR i WAW

Les dependències veritables (dependències RAW) tenen lloc quan una instrucció llegeix un valor d'una posició d'emmagatzematge que una instrucció més vella escriu. Pot donar-se un problema si la instrucció jove llegeix el valor abans que la instrucció vella l'hagi escrit.

```
ADD R0, R1, R2
ADD R3, R0, R1
```

Figura 16: Exemple de dependència veritable

Aquesta situació sí pot ocasionar un risc, degut únicament al bucle hardware entre l'etapa WB i l'etapa DR. Cal notar que només es poden produir riscos de dades deguts a registres i no es poden produir riscos de dades deguts a memòria. Això es justifica d'igual forma que la no problemàtica de les dependències WAR i WAW: un Store sempre escriu abans que llegeixi un Load més jove, un Load sempre llegeix abans que escrigui un Store més jove i un Store sempre escriu abans que escrigui un Store més jove.

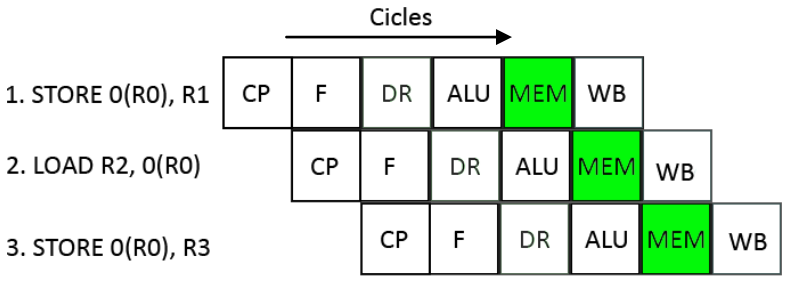


Figura 17: Inexistència de riscos RAW per culpa de la memòria

El risc serà detectat a l'etapa DR, quan es descodifiquin els identificadors dels registres font i es comparin amb l'identificador del registre destí a les etapes posteriors. Per a solucionar aquest tipus de risc, es bloquejarà la interpretació de les dues instruccions posteriors, que estan a les etapes PC i Fetch. A les tres etapes següents, se les injecta una instrucció NOP mentre es detecti risc per a evitar que s'actualitzi l'estat del processador.

Reducció dels cicles perduts per riscs de dades: Curtcircuits

Els curtcircuits són un mecanisme per a obtenir una dada que ja es troba calculada en algun punt del pipeline però que encara no ha estat escrita al banc de registres, mitjançant un camí auxiliar des d'una etapa productora a l'etapa que consumidora. Cal, a més, un multiplexor per a

seleccionar si la dada arriba pel camí habitual o si utilitza el curtcircuit.

S'afegiran curtcircuits entre les etapes WB i DR, MEM i DR i ALU i DR. Això implica que:

- La interpretació d'una instrucció aritmeticològica seguida de qualsevol altra instrucció que ocasioni una dependència RAW no comporta pèrdua de cicles per culpa d'aquesta dependència (es produeix el resultat a l'etapa ALU i es consumeix a l'etapa DR).
- La interpretació d'una instrucció Load seguida de qualsevol altra instrucció que ocasioni una dependència RAW només comporta la pèrdua d'un cicle per culpa d'aquesta dependència (es produeix el resultat a l'etapa MEM i es consumeix a l'etapa DR).

2.5.2 Riscos de seqüenciament

Un risc de seqüenciament és la detecció de la possibilitat d'interpretar una sèrie d'instruccions diferent que la especificada pel programador.

Això és degut a que, en la interpretació d'instruccions de salt, el PC pot ser actualitzat a l'etapa WB. Com a l'arribar a l'etapa WB ja s'han començat a interpretar 4 instruccions, l'estat del processador pot haver estat alterat de forma diferent a la intencionada.

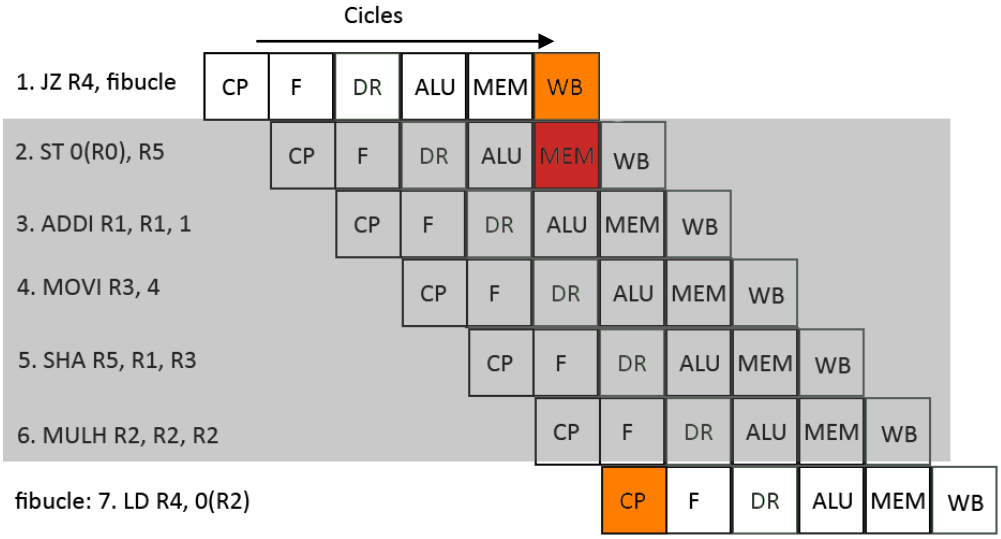


Figura 18: Modificació de l'estat del processador per culpa d'un risc de seqüenciament

El risc serà detectat a l'etapa DR, quan es descodifiqui el tipus d'instrucció i se sàpiga que és una instrucció de ruptura del seqüenciament. Per a solucionar aquest tipus de risc, es bloquejarà la interpretació de les dues instruccions posteriors, que estan a les etapes PC i Fetch. A les tres etapes següents, se les injecta una instrucció nop mentre es detecti risc per a evitar que s'actualitzi l'estat del processador.

Reducció dels cicles perduts per riscs de seqüenciament: Predictor estàtic de salts

Un predictor de salt és un circuit que intenta saber si un salt es pren o no abans que es decideixi. Un predictor estàtic treballa només amb la informació que aporta la instrucció de salt i, en particular, no té en compte la història dinàmica de l'execució del codi.

Aquest processador implementa un tipus de predictor que es fixa en l'adreça de salt. Salta si l'adreça destí és menor que l'actual adreça del PC i no ho fa si l'adreça destí és major. La raó d'aquesta tria és l'afavoriment del rendiment dels bucles. Els bucles acostumen a tenir una estructura similar a: comprovar la condició del bucle, executar el cos del bucle i tornar a la comprovació. Aquesta tornada és un salt endarrere (menor que el PC actual), ja que el destí del salt (la instrucció de comprovació) és una instrucció anterior. Com que els bucles s'han d'executar moltes vegades, aquest predictor encertarà durant totes les iteracions menys la última, obtenint una pèrdua de cicles per culpa d'aquest risc molt baixa.

Amb aquest predictor, si es prediu no seguir en seqüència i s'encerta, es perd 1 cicle mentre que, si es falla la predicció, se'n perden 2. Si es prediu seguir en seqüència i s'encerta, no es perd cap cicle mentre que, si es falla la predicció, se'n perden 2. Sense predictor, com el pipeline esperava a que el comptador de programa s'escrivís, es perdien sempre 6 cicles.

2.5.3 Riscos estructurals

Un risc estructural es dóna quan dues instruccions volen utilitzar un recurs en el mateix cicle.
En aquesta implementació, l'únic recurs compartit per dues etapes és la memòria, ja que no està separada en memòria de dades i memòria d'instruccions i només utilitza un port d'accés. Hi haurà un risc estructural en el moment en què una instrucció Load o Store vulgui accedir a memòria ja que, a la vegada, una altra instrucció estarà a l'etapa Fetch i també hi voldrà accedir.

Es decideix no solucionar el risc implementant memòries cache o dos ports per a accedir a memòria per a poder estudiar el seu efecte, que es considera interessant i no es tractava a l'assignatura AC2. El risc serà detectat a l'etapa DR, quan es descodifiqui el tipus d'instrucció i se sàpiga que accedeix a memòria. Per a solucionar aquest tipus de risc, quan s'arribi a l'etapa MEM, es bloquejarà la interpretació de les instruccions que estan a les etapes PC, Fetch i DR, mentre que s'injectarà una NOP a les etapes ALU i WB.

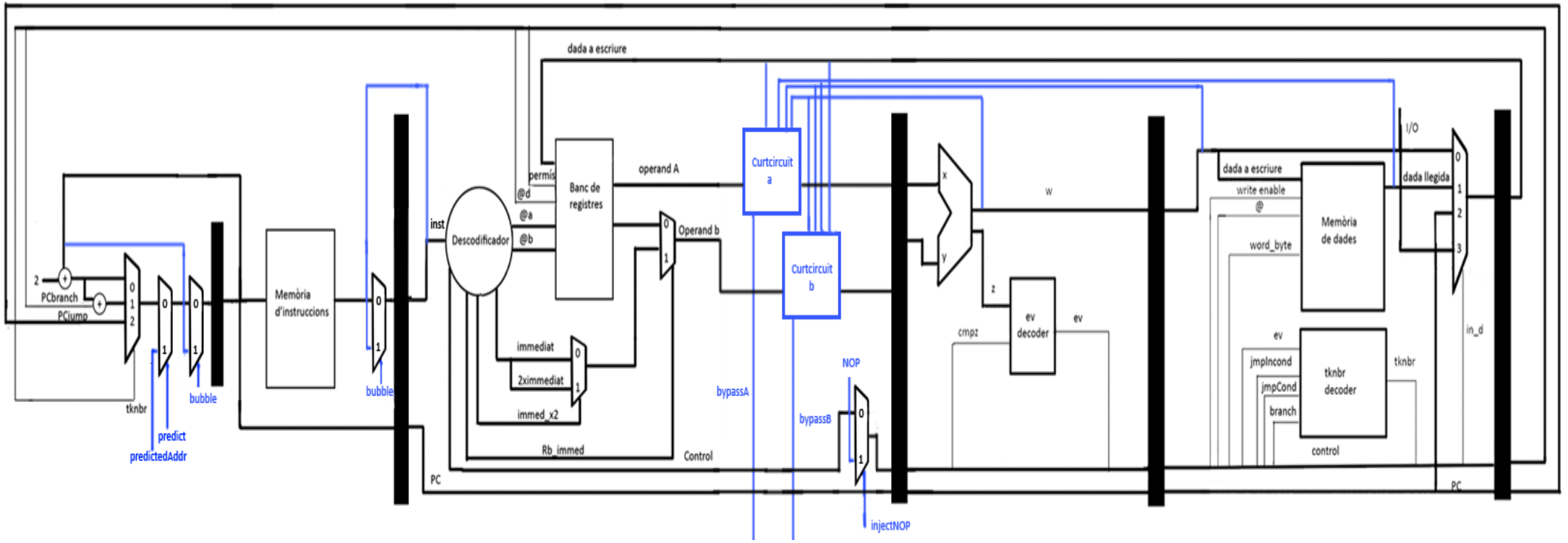


Figura 19: Pipeline amb curtcircuits i predictor

3. Implementació del processador

En aquest capítol s'explicarà amb força detall com s'ha implementat el projecte: senyals i mòduls usats, decisions preses per a fer que el projecte funcionés i exemples de com treballa el processador. El capítol segueix l'ordre cronològic de la implementació: el mode sistema, la segmentació i els curtcircuits i predictor de salt.

3.1 Mode sistema

3.1.1 Banc de registres

Comencem implementant els registres de sistema. Originalment, el banc de registres està implementat com un mòdul que rep com a entrades el rellotge, dues adreces de lectura, una d'escriptura, un permís d'escriptura i una dada per escriure, i té com a sortides els dos valors llegits.

Els registres estan definits com un senyal de tipus array de 8 posicions on cadascuna hi ha un std_logic_vector de 16 bits. Simplement afegint un altre senyal igual tenim l'estructura de dades creada.

El mòdul necessita rebre nous senyals per a gestionar el mode sistema. D'entrada, són necessaris dos bits que indiquin si s'està llegint o escrivint en mode usuari o sistema. Només són necessaris dos bits perquè es llegeix o s'escriu només un registre alhora, i a les instruccions sempre està codificat als mateixos bits. Amb aquests bits, podem utilitzar un multiplexor de l'estil:

```
with a_sys select
a <= sys_reg(to_integer(unsigned(addr_a))) when '1',
     reg(to_integer(unsigned(addr_a)))  when others;
```

per a llegir valors del banc. En aquest codi, a és un valor de sortida de 16 bits, corresponent a la lectura d'un registre font, addr_a és l'adreça del registre a llegir i reg i sys_reg són els dos bancs de registres. El senyal a_sys controla el multiplexor.

Per a escriure en un registre, es feia servir un process per a controlar que l'escriptura estigués sincronitzada amb el rellotge. Aprofitem aquest no només per la instrucció wrs, sinó també per a guardar la informació necessària. El fragment de codi següent ho aconsegueix:

```
if (sys_sig = '1') then
    sys_reg(0) <= sys_reg(7);
    sys_reg(1) <= pc;
    sys_reg(2) <= x"000"&event_id;
    sys_reg(7) <= x"0002";
end if;
```

Es guarda el que cal en els registres, però cal afegir els senyals *sys_sig* i *event_id*. El primer ens indicarà que hem entrat al mode sistema i que, per tant, cal realitzar totes les accions que es veuen en el codi. El segon, és un senyal que indicarà quin esdeveniment s'ha produït. Aquestes senyals les ha de generar un nou mòdul, un controlador d'interrupcions, que ara s'explicarà.

Abans, però, l'últim detall de la implementació del nou banc de registres. En el moment que s'entra al mode sistema, es guarda la paraula d'estat al registre S0. Aquesta, s'ha de restaurar quan s'executa la instrucció de retorn d'interrupció reti. Per tant, s'afegirà un senyal d'entrada que indiqui que s'està executant un reti i que cal restaurar S7. La informació que s'està interpretant una instrucció reti es coneix en el moment de descodificació, pel que serà el descodificador l'encarregat de generar el senyal. Com s'està escrivint en un registre, també s'afegeix aquesta acció dins del process descrit anteriorment.

3.1.2 Controlador d'interrupcions

Amb tots els senyals afegits, cal un control de la seva lògica. No només això, sinó que cal un lloc on es gestionin les diferents interrupcions que poden succeir alhora i una interfície que rebi pròpiament les interrupcions. És per això que s'implementa aquest nou mòdul.

El mòdul ha de rebre informació sobre les diferents peticions d'interrupció que puguin succeir, sobre si el programa vol entrar a mode sistema (instrucció calls) i sobre si les interrupcions estan habilitades o no. Amb això, el mòdul és capaç d'enviar resposta als altres mòduls del processador sobre que efectivament s'ha entrat a mode sistema, el tractament de la interrupció i l'esdeveniment que s'està tractant.

S'entra al mode sistema quan es rep una interrupció i aquestes estan habilitades o quan s'executa una instrucció calls:

```
sys_sig <= '1' when (inte = '1' and intr /= x"0") or calls = '1' else '0';
```

En el codi anterior, *inte* indica la permissió d'interrupcions (enable) i *intr* indica una petició d'interrupció (request). *calls* és un senyal generat pel descodificador a l'interpretar una instrucció calls.

El fet d'indicar a un dispositiu que s'està tractant la interrupció i de notificar a la resta del processador que s'està tractant una interrupció van lligats en concepte. A la pràctica s'usa un codificador amb prioritat, que rep el mapa de bits de peticions (*intr*) i té com a sortida l'identificador d'esdeveniment, que es pot fer servir a la resta del processador com a informació. El mòdul utilitza aquest senyal i el descodifica en un altre mapa de bits de resposta (acknowledgement) anomenat *inta*. Cada dispositiu rep un d'aquests bits, i sap si la seva petició s'està tractant (el bit val '1') o no (el bit val '0'). La següent figura il·lustra la petició d'interrupció per part de dos dispositius i la gestió del mòdul de control:

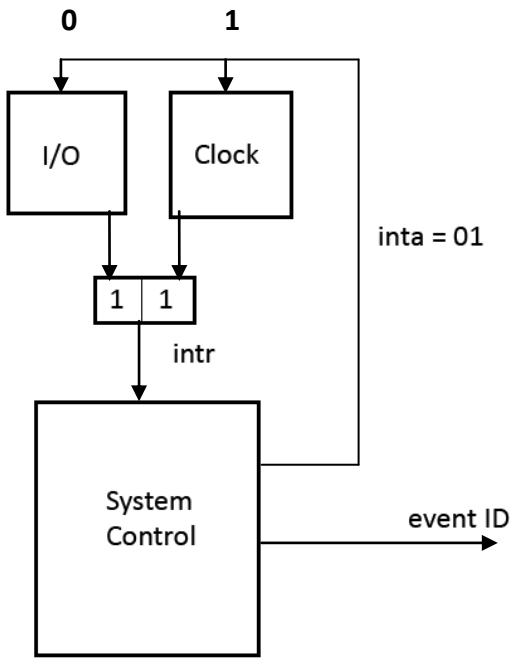


Figura 20: Petició de dues interrupcions alhora

Veiem com tant el rellotge com un dispositiu d'entrada sortida (els botons) demanen volen interrompre al processador. Cadascú envia un bit, i el controlador els rep com un vector. Aquest vector és codificat en dos senyals diferents: una, *event_id*, que s'envia a altres mòduls del processador. L'altra, es divideix en diferents bits que s'envien als dispositius. En aquest cas, com el botó i el rellotge tenen una petició alhora i no hi ha cap altre petició, el valor *intr* és 3, o 11 en binari (el rellotge es mapeja a la posició '0' del vector i, els botons, a la '1'). El rellotge té més preferència que els botons, pel que el valor 3 té com a sortides 10 per *event_id* (10 és l'identificador triat pel rellotge) i b"01" pels dispositius, el que significa que el ack dels botons val '0' i, el del rellotge, '1'. Per tant, es tracta la interrupció de rellotge i no la d'entrada/sortida.

Amb aquesta estratègia de disseny, es pot ampliar el número de dispositius sense haver de modificar els senyals que envien o reben els dispositius; només cal modificar el codificador i els senyals del controlador d'interrupcions.

3.1.3 Modificació dels dispositius

El processador original té un mòdul, controladoresIO, que gestiona la lectura i escriptura dels LED, interruptors i botons de la placa al fer servir les instruccions IN i OUT. Ara s'implementarà la possibilitat d'enviar interrupcions, així com un altre dispositiu (un rellotge) per a verificar el funcionament del codificador de prioritat.

Per a habilitar les interrupcions en els dispositius d'entrada, s'opta per un graf d'estats.

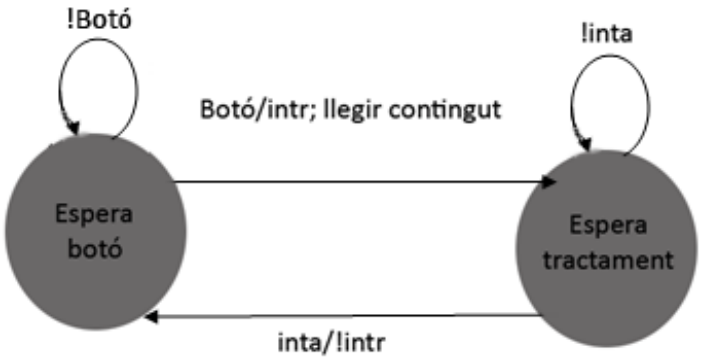


Figura 21: Graf del tractament de la interrupció de botons

El que fa el graf és esperar que es premi un botó en el cas dels keys o que es mogui un interruptor en el cas dels switches. Llavors, es llegeix el contingut, s'envia la petició al controlador d'interrupcions i es canvia al segon estat. En aquest, s'espera a rebre el senyal de tractament de la interrupció per part del processador. Quan es rep, es desactiva la petició i es torna al primer estat. La única diferència entre els grafs dels botons i dels interruptors és el número de bits que usa la variable que llegeix el contingut, ja que la placa té 10 switches i 4 keys.

La implementació del temporitzador també fa servir un graf d'estats similar al d'entrada/sortida. El que provoca la transició del primer al segon estat, però, és que hagi passat una certa quantitat de temps. Això s'aconsegueix amb el control d'overflow d'un comptador, implementat de la següent forma:

```
if(rising_edge(clk))then
   clk_counter <= clk_counter + "1";
end if;
if (clk_counter = x"FFFFF") then
   estat <= "10";
   intr <= '1';
end if;
```

Quan el senyal *clk_counter* desborda, vol dir que ha passat el temps que es volia, i s'envia la interrupció. Modificant la mida del comptador i fins a quin valor es vol arribar, es pot controlar la freqüència de les interrupcions.

S'encapsula el controlador d'entrada/sortida i el temporitzador en un únic mòdul, anomenat dispositius. Això es fa pensant en la modularitat del processador a l'hora d'ampliar el projecte. Aquest mòdul, a més, permet que no sigui el controlador d'interrupcions el que s'encarregui de distribuir els ack d'interrupcions. Ara, el controlador d'interrupcions rep un vector de peticions i envia un vector de respostes. El controlador d'interrupcions s'encarrega de gestionar el vector de peticions i, el mòdul dispositius, el de respostes.

3.1.4 Prova del funcionament de les interrupcions

El codi següent encén els LED indicant, en binari, el número del botó que s'ha premut. Per exemple, si es pitja el quart botó (KEY 3), s'encén el tercer LED (fent el número "100", 4 en binari) o, si es pitja el tercer botó (KEY2), s'encenen el primer i segon LED (fent el número "11", 3 en binari).

Les primeres instruccions del codi escriuen a S5 l'adreça de la RSI, que s'executarà quan es rebi una interrupció. Després, el codi es queda en un bucle infinit, esperant-les. A continuació, apareix el codi de la RSI: es comprova que és una interrupció de botons (comprovant el seu identificador, que és 9) i, si ho és, executa instruccions per a obtenir el resultat explicat anteriorment. Si no és una interrupció de botons, se surt de la RSI.

```
        MOVI R1, 3
        MOVI R2, 14
        SHL R1, R1, R2
        MOVI R2, 3
        MOVI R3, 5
        SHL R2, R2, R3
        OR R1, R1, R2

        LD R2, 0(R1)
        WRS S5, R2

        MOVI R0, 0
        EI
while1:
        BZ R0, while1

.org 96
.word 0xC080; @RSI

.org 128
rsi:
        GETIID R1
        ADDI R1, R1, -9
        BNZ R1, firsi
        IN R2, 9
        ADDI R2, R2, -3
        BZ else
        OUT 6, R2
        JMP firsi
else:
        MOVI R2, 3
        OUT 6, R2
firsi:
        RETI
```

Per tant, mentre no es premi un botó no passarà res, encara que arribin les interrupcions de rellotge. Si arriba una interrupció de rellotge i una de botó alhora, tampoc passarà res, doncs la de rellotge té més prioritat. En el moment en què només la interrupció de botons estigui pendent, s'encendran els LED.

En aquesta imatge es veu com, polsant el segon botó, s'encén el segon LED (indicant "10", 2 en binari).

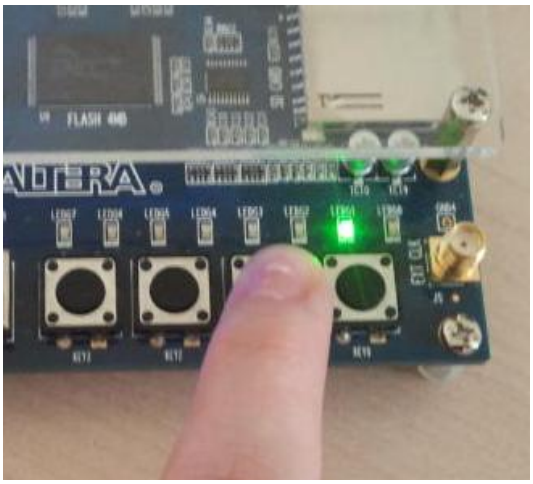


Figura 22: Funcionament de les interrupcions

3.2 Implementació del pipeline

Es decideix implementar el processador fent servir els següents mòduls:

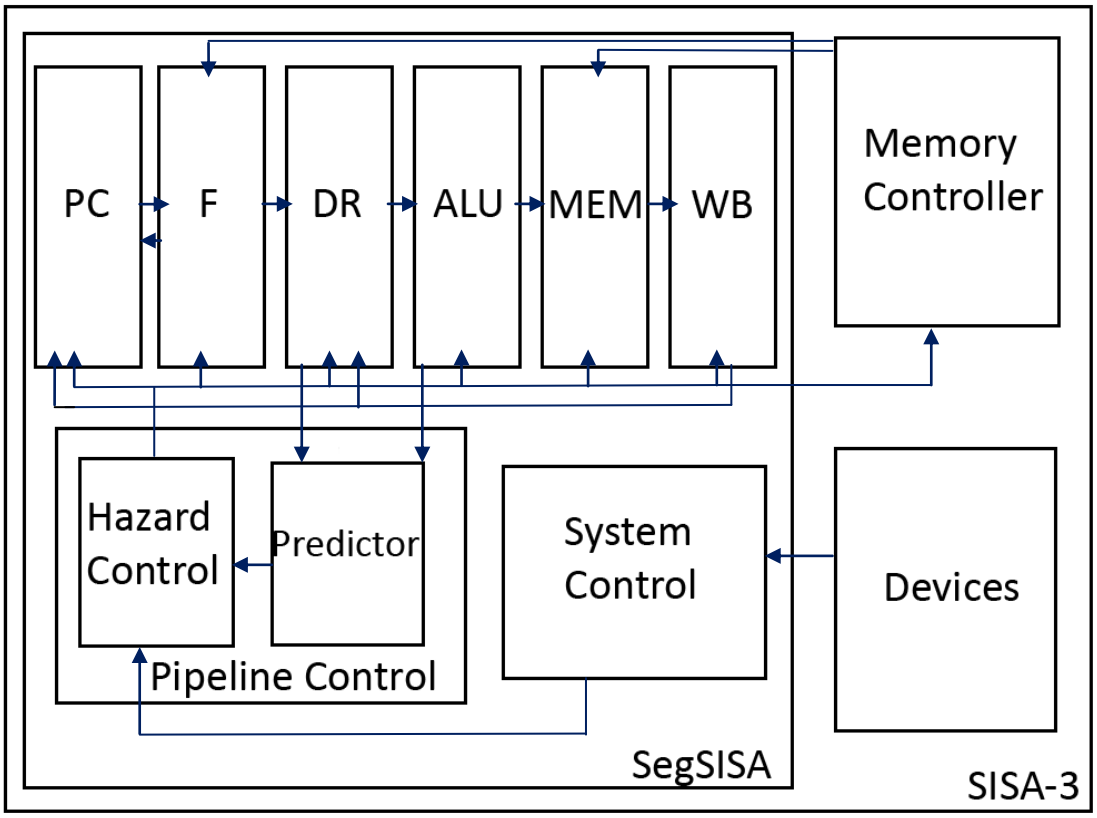


Figura 23: Esquema de mòduls del processador

Observem que hi ha un mòdul exterior, SISA-3, que es comunica amb els pins de la placa. Aquest mòdul instancia el controlador de memòria, els dispositius i un mòdul anomenat SegmentedSISA. A la seva vegada, el SegmentedSISA instancia un mòdul per etapa, el controlador d'interrupcions i el controlador del pipeline. En aquest apartat s'explica com estan implementats els mòduls que formen el pipeline i la primera versió del controlador, que gestiona els tres tipus de riscos.

3.2.1 Etapes del pipeline

Program Counter

L'etapa instancia un registre de desacoblament, on es guardarà el PC. La lògica de l'etapa és l'entrada d'aquest registre, gestionada per dos multiplexors. El primer, controlat pel senyal *tknbr*, indica quin tipus de seqüenciament s'utilitza. El segon, controlat pel senyal *bubblePC*, controla si es carrega un nou PC o l'etapa fa una bombolla.

En el següent fragment de codi, es veuen els multiplexors i la instància del registre. El senyal *newPCOut* és la sortida de l'etapa.

```
ComptadorPrograma : RegPC
   port map(
      clk      => clk,
      boot     => boot,
      D     => bus_DPC,
      Q     => PC
   );

bus_DPC <= MuxOut  when bubblePC = '0' or (bubblePC = '1' and tknbr = "10") else
           PC when bubblePC = '1' and tknbr /= "10";


with tknbr select
MuxOut <= (PCbranch+2) + (RelShiftBranch(7)&RelShiftBranch(7)&RelShiftBranch(7)
&RelShiftBranch(7)&RelShiftBranch(7)&RelShiftBranch(7)&RelShiftBranch(7)
&RelShiftBranch(7 downto 0)&'0') when "01", --Branch

PCjmp+2 when "10", --Jmp
PC+2  when others; -- seq implicit.
```

Fetch

Aquesta etapa rep el PC guardat a l'etapa anterior, i tindrà com a sortida la instrucció portada de memòria. Instancia un parell de registres de desacoblament per a guardar la instrucció i el PC. Cal propagar el PC per a poder calcular adreces de salt en etapes posteriors.

En el següent fragment de codi es veu la instància dels registres de desacoblament. El senyal *addrToMemory* s'envia al controlador del pipeline, i es rep *instFromMemory* del controlador de memòria. El senyal *instrOut* i *PCOut* són els que es passen a la següent etapa. El multiplexor controlat pel senyal *bubbleFetch* serveix per a fer una bombolla a l'etapa: es preserva la instrucció llegida del cicle anterior.

```
DesacoblamentF_D : RegsFetch_Decode
   port map(
      clk      => clk,
      boot     => boot,
      Dinstr   => bus_Dinstr,
      DPC      => PCin,
      Qinstr   => bus_instrOut,
      QPC      => PCOut
   );

addrToMemory <= PCin;
instrOut <= bus_instrOut;

with bubbleFetch select --gestio del risc estructural
    bus_Dinstr <= instFromMemory when '0',
    bus_instrOut when others; --bombolla
```

Decode & Read

Aquesta etapa instancia el banc de registres, el descodificador i una sèrie de registres de desacoblament que guarden senyals de control.

Degut a la implementació del banc de registres, no es pot escriure i llegir en un registre en el mateix cicle i que el valor llegit correspongui al valor escrit.

La lògica de l'etapa llegeix el valor dels operands font: el valor d'un registre per l'operand a i tria si un registre o un immediat per l'operand b. Això s'aconsegueix amb aquest parell de multiplexors:

```
with bus_immed_x2 select
   actual_immed <= bus_immed when '0',
   bus_immed(14 downto 0) & '0' when others;

with bus_rb_inm select
   bus_DopB <= bus_opbregfile when '1',
               actual_immed when others;
```

El primer selecciona entre l'immediat i l'immediat multiplicat per dos i, el segon, entre l'immediat resultant i el valor llegit del banc de registres.

L'etapa genera una sèrie de senyals de control generals i una sèrie de senyals de control per a components usats en etapes posteriors. Aquests segons, s'agrupen per etapes en vectors de bits, de forma que s'afavoreix la modularitat del desacoblament: es veu que els senyals necessaris travessen les etapes, però no es veu quins senyals en concret. Cada etapa pot fer servir aquests vectors per a controlar components o afegir senyals per a les següents etapes.

A la següent figura es mostra com s'agrupen els senyals i com es propaguen en els diferents registres de desacoblament, basat en l'arquitectura MIPS[7]:

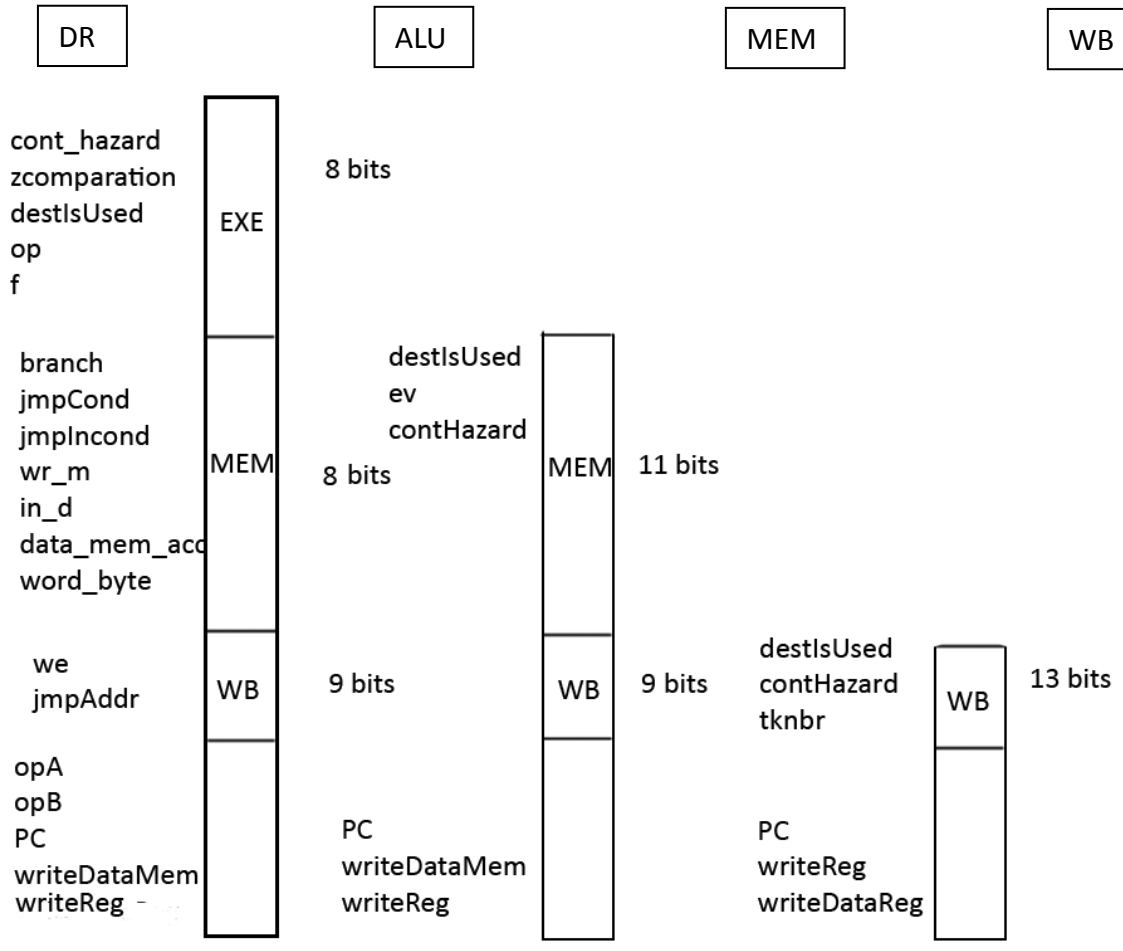


Figura 24: Propagació dels senyals de control

A cada etapa, a l'entrada es mostra amb quants bits entra el senyal de control i, a l'entrada de la següent etapa, amb quants surt. Per exemple, el senyal de control de l'etapa Write-back entra a l'etapa Memory amb 9 bits. A l'etapa Memory se li afegeixen els bits de *tknbr*, *destIsUsed* i *cont_hazard*, pel que entra a l'etapa Write-Back amb 13 bits.

Els senyals que es veuen propagant-se per la part inferior de la figura i que no tenen etiquetada cap etapa són els senyals de control generals.

Els senyals usats són:

- Cont_hazard: Indica si hi ha un risc de seqüenciament a l'etapa.
- Zcomparation: Indica si el branch o el jump fan una comparació amb zero.
- destIsUsed: Indica si la instrucció fa servir un operand destí.
- Op: Operació de l'ALU.
- F: Funció de l'ALU.
- Branch: Indica que és una instrucció branch.
- Jmpcond: Indica que és una instrucció jmp condicional.
- JmpIncond: Indica que és una instrucció jmp incondicional.
- Wr_m: Permís d'escriptura a memòria.
- In_d: Senyal de selecció de la dada a escriure en el registre destí.
- Data_mem_access: Indica que la instrucció és un load o un store.
- Word_byte: Indica la granularitat de l'accés a memòria de dades.
- We: Permís d'escriptura a registres.
- jmpAddr: Adreça on cal saltar en una instrucció jmp.
- Ev: Resultat d'avaluar la condició de salt.
- Tknbr: Senyal de selecció del seqüenciament.
- opA: Operand a.
- opB: Operand b.
- PC: Comptador de Programa.
- writedataMem: Dada a escriure a memòria de dades.
- writeRegister: Adreça del registre destí.

L'etapa rep 3 senyals causades per riscos: *salta*, *branchrisc* i *injectNOP*. Les tres són generades pel control del pipeline, i es discutirà la seva lògica a l'apartat corresponent.

El senyal *salta* indica que el processador no segueix el seqüenciament implícit, pel que enlloc de descodificar la instrucció buscada per l'etapa Fetch, es descodifica una NOP.

```
with salta select
   bus_ir <= fetchedInstr when '0',
             x"FFC0" when others;  --NOP
```

El senyal *branchrisc* indica que cal restar-li 2 al PC que s'ha propagat. Degut a la implementació, a l'etapa PC se suma 2 abans que es detecti el risc, pel que s'està retenint un PC erroni, que es propagarà quan se solucioni el risc. Això no és important a no ser que s'executi una instrucció branch, que necessita el PC per a calcular l'adreça efectiva de salt.

```
with branchrisc select
      bus_PC <= PCin - 2 when '1',
                PCin  when others;
```

El senyal *injectNOP* es fa servir per a modificar alguns bits dels senyals de control dels components, com per exemple permisos d'escriptura o indicar que no s'està interpretant una instrucció de salt. En el fragment adjunt es mostren, a mode d'exemple, com es modifiquen dues senyals: la que indica si hi ha un risc o no de seqüenciament i la que indica si s'usa o no un registre destí. Si el senyal *injectNOP* val '1', aquestes senyals han de valer '0'. Primerament, perquè no han de modificar l'estat del processador (els permisos d'escriptura) i, a més, no han de causar riscos innecessaris (*control_hazard* indica que hi ha una instrucció de salt executant-se, el que pot causar risc).

```
with injectNOP select
bus_DsenyalsControlEXE(7) <= '0' when '1',
   bus_controlhazard when others;

with injectNOP select
   bus_DsenyalsControlEXE(5) <= '0' when '1',
   bus_destIsUsed when others;
```

Execution

Aquesta etapa instancia l'ALU i una sèrie de registres per a emmagatzemar senyals de control. L'etapa s'encarrega de distribuir els senyals de control dels components que rep per a assignar totes les entrades de l'ALU.

S'ha triat un disseny modular per l'ALU separat per tipus d'operacions. Al tenir quatre codis d'operació, l'ALU està formada per quatre components que reben cadascun x, y i f i calculen un resultat. Cada mòdul (logicoaritmètic, comparació, extensió aritmètica i mòdul de moviment i memòria) realitza una operació depenent del codi *f* que rep. Després, amb el senyal *op*, l'ALU selecciona quina de les quatre sortides utilitzar com a resultat de la instrucció i quina de les quatre sortides cal comparar amb zero per a activar o desactivar el bit *z*. El bit *z* val '1' si la comparació amb zero és certa. L'ALU fa servir dues senyals per a escollir quin resultat d'operació posar al port *w*: un senyal *op* de 2 bits i un senyal *f* de 3 bits.

A la següent taula es pot veure com es distingeixen les operacions gràcies a aquests senyals:

f			**OP**			
f2	f1	f0	0 0	0 1	1 0	1 1
0	0	0	AND	CMPLT	MUL	MOVI
0	0	1	OR	CMPLE	MULH	MOVHI
0	1	0	XOR	-	MULHU	LD
0	1	1	NOT	CMPEQ	-	LDB
1	0	0	ADD	CMPLTU	DIV	ST
1	0	1	SUB	CMPEU	DIVU	STB
1	1	0	SHA	-	-	-
1	1	1	SHL	CMPS	ADDI	-

Tot i que les instruccions LB, LDB, ST, STB i ADDI fan la mateixa operació que la instrucció ADD (sumar els operands x i y), s'ha preferit posar-les explícitament per a controlar millor quines senyals passen pels busos i què s'està executant en cada moment. La instrucció CMPS es tradueix com "Comparació en un salt" i és una operació interna que s'ha definit per a implementar els salts. Aquesta operació compara l'operand b amb "0" i deixa passar l'operand a, que és la direcció absoluta o relativa a la que es vol saltar.

Per a implementar-la, s'afegeix una sortida addicional al mòdul de comparacions anomenada *direcciojmp*, que valdrà l'operand a quan la instrucció que s'està executant és un CMPS. A més, en el mòdul afegim que si *f* = "111", l'operand b es compari amb "0" i deixi el resultat d'aquesta comparació en el bus que conté el resultat d'aquest mòdul.

En el mòdul de l'ALU s'afegeix un multiplexor posterior al que seleccionava entre busos de resultats segons el senyal *op*. Aquest multiplexor, controlat per si la operació és un CMPS o no, selecciona entre el resultat del multiplexor anterior i el senyal *direcciojmp*. D'aquesta manera, pel port *w* surt la direcció absoluta o relativa a saltar si la operació que fa l'ALU és un CMPS i, si no, surt el resultat de la operació pertinent.

Amb el bit *z* que surt de l'ALU i els senyals de control de components de l'etapa, es pot calcular el senyal *ev,* que es fa servir a l'etapa de memòria per a decidir el salt. El senyal es calcula de la següent forma:

```
bus_ev <= '1' when (senyalsControlEXE(6) = '0' and bus_z = '0') or
                   (senyalsControlEXE(6) = '1' and bus_z = '1')
          else '0' ;
```

El bit senyalsControlExe[6] indica si la comparació es fa amb zero o no. Per tant, *ev* val '1' si no es fa una comparació amb zero i la comparació amb zero dóna falsa (*z*='0') o si es fa una comparació amb zero i la comparació amb zero dóna cert (*z*='1').

A part de fer l'operació a l'ALU i calcular el senyal *ev*, l'etapa col·loca als senyals de control dels components de memòria els bits *destIsUsed*, *ev* i *cont_hazard*.

Memory

Aquesta etapa instancia els registres de desacoblament corresponent als senyals de control de l'etapa Write-back, les adreces de destí de salt, l'adreça del registre destí i la dada a escriure al banc de registres.

L'etapa es connecta amb el controlador de memòria mitjançant els senyals de permís d'escriptura, granularitat de l'accés i dada a escriure, i n'obté la dada llegida. El senyal d'adreça s'envia al controlador del pipeline. Si la instrucció executada és un Load o un Store, s'envia un avís al controlador del pipeline per a que gestioni el risc estructural.

```
data_mem_access <= senyalsControlMEM(1);
```

Es fa servir el senyal *in_d* per a seleccionar la font de la dada que s'escriurà al registre destí. Aquesta pot ser el resultat de l'ALU, la dada llegida de memòria, el PC o una dada llegida dels

dispositius d'entrada/sortida. En el codi següent, *bus_Dd* és l'entrada al registre de desacoblament que conté d, la dada a escriure en el registre.

```
with senyalsControlMEM(4 downto 3) select  --in_d
   bus_Dd <= w when "00",
             datard_m when "01",
             PCin when "10",
             rd_io when others;
```

L'etapa de memòria col·loca al bus de senyals de control de l'etapa Write-back els senyals *destIsUsed*, *cont_hazard* i *tknbr*, que s'obté en aquesta etapa.

El senyal *tknbr* serveix per a seleccionar el tipus de seqüenciament: implícit, jump o branch, i es calcula de la següent manera:

```
bus_DsenyalsControlWB(10 downto 9) <=
   "01" when senyalsControlMEM(8) = '1' and senyalsControlMEM(9) = '1' else
   "10" when (senyalsControlMEM(7) = '1' and senyalsControlMEM(9) = '1')
   or senyalsControlMEM(6) = '1' else
   "00";
```

senyalsControlWB[10:9] és el senyal *tknbr* a l'etapa Write-Back. *senyalsControlMEM*[8] indica si la instrucció és un branch, *senyalsControlMEM*[7] indica si és un jump condicional, *senyalsControlMEM*[6] indica si és un jump incondicional i, per últim, *senyalsControlMEM*[9] és *ev*.

Per tant, *tknbr* val "1" (branch) si la instrucció és un branch i l'avaluació és positiva, "2" (jump) si el salt és condicional i l'avaluació és positiva o si és un salt incondicional i "0" (seqüenciament implícit) en qualsevol altre cas.

Write-Back

Aquesta etapa no instancia registres, simplement alimenta l'etapa DR amb els senyals del banc de registres i l'etapa PC amb el senyal del control del multiplexor. A més, envia els senyals *destIsUsed* i *cont_hazard* al controlador del pipeline per a gestionar riscos.

```
destIsUsed <= senyalsControlWB(11);
controlHazard <= senyalsControlWB(12);

we_out <= senyalsControlWB(8);

tknbr_out   <= senyalsControlWB(10 downto 9);

d_out       <= d_in;
addr_d_out      <= addr_d_in;
 jmpAddr_out       <= senyalsControlWB(7 downto 0);
PCbranch_out    <= PCbranch_in;
PCjmp_out       <= PCjmp_in;
```

3.2.2 Controlador del pipeline

Aquest mòdul ha de generar els tres senyals que aturen les tres primeres etapes quan hi ha qualsevol tipus de risc, el senyal *salta* i el senyal *branchrisc*. A més, ha de gestionar el risc estructural de memòria, ja que tant l'etapa de Fetch com l'etapa de Memòria li envien les adreces d'accés.

Per a controlar els riscos de dades, el controlador necessita les adreces dels operands font a l'etapa DR i les adreces dels operands destí a les 3 etapes posteriors. S'activaran els senyals d'aturar les etapes quan un dels dos operands coincideixi amb algun dels registres destí de les etapes posteriors.

A més, és necessari un bit de validesa que indiqui si aquella instrucció realment farà servir el registre destí. Això és necessari per a evitar situacions com la que es veuen en aquesta figura:

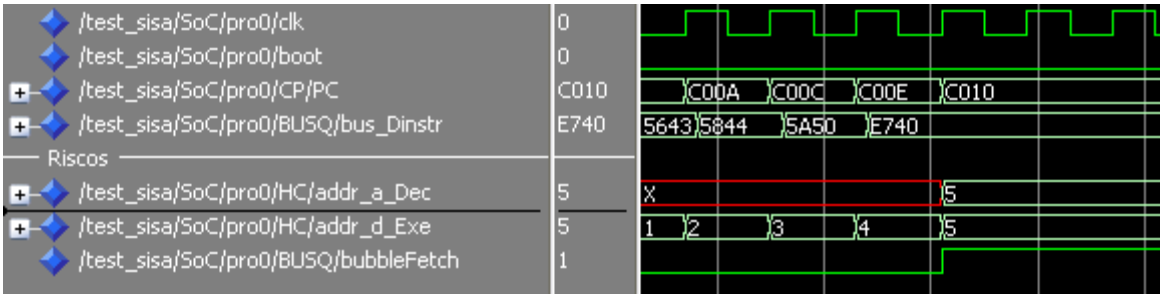


Figura 25: Necessitat del bit d'ús del registre destí

Si un senyal no se sobreescriu, el seu valor perdura al llarg del temps. En la situació de la figura, s'executa un MOVI amb registre destí 5 (MOVI R5, 80: 0x5A50) i després un ST que fa servir el registre 5 (ST 0(R5), R3: 0xE740). Com que l'Store no modifica el valor del registre destí, la comparació que a l'etapa DR hi ha un registre font que coincideix amb algun registre destí serà

sempre certa, i sempre hi haurà un risc. Per tant, cal afegir a la comparació una condició que indiqui que s'usa el registre destí, senyal que es descodifica a l'etapa DR i es va propagant.

Per a controlar els riscos estructurals, cal un senyal que indiqui si a l'etapa de memòria hi ha una instrucció Load o Store. En cas afirmatiu, s'aturen les primeres 3 etapes i s'accedeix a memòria de dades. Un multiplexor controlat per aquest senyal envia l'adreça a memòria: si no hi ha Load o Store s'accedeix amb l'adreça de l'etapa Fetch; si n'hi ha, s'accedeix amb l'adreça de l'etapa Memòria.

```
with data_mem_access_Mem select
    bus_addr_m <= addr_m_memory when '1',
                  addr_m_fetch when others;
```

Per a controlar els riscos de seqüenciament, només cal un senyal que digui si s'està interpretant una instrucció de salt a l'etapa. En cas afirmatiu, s'aturarà el pipeline, ja que fins l'etapa WB no s'escriu el nou PC.

La lògica de les senyals que aturen el pipeline queda així:

```
bus_bubblePC <= '1' when (
    (addr_a_Dec = addr_d_Exe and destIsUsedExe = '1' )  or
    (addr_b_Dec = addr_d_Exe  and destIsUsedExe = '1') or
    (addr_a_Dec = addr_d_Mem  and destIsUsedMem = '1') or
    (addr_b_Dec = addr_d_Mem  and destIsUsedMem = '1' ) or
    (addr_a_Dec = addr_d_WB and destIsUsedWB = '1' )   or
    (addr_b_Dec = addr_d_WB and destIsUsedWB = '1' )   or
    data_mem_access_Mem = '1' or
    controlHazardExe = '1' or controlHazardMem = '1' or  controlHazardWB = '1') else
    '0';
```

El senyal *salta* indica a l'etapa DR que el programa trenca el seqüenciament implícit. Això passa si hi ha una instrucció de salt a l'etapa WB i *tknbr*, el senyal que controla el multiplexor del PC, té un valor diferent de "0". Com que aquesta informació se sap en el cicle c però s'ha d'aplicar al cicle c+1, cal un process per a endarrerir aquesta informació.

```
process(clk, controlHazardWB, tknbr)
begin
   if (rising_edge(clk)) then
      if (controlHazardWB = '1' and tknbr /= "00") then
         salta <= '1';
      else salta <= '0';
      end if;
   end if;
end process;
```

El senyal *branchrisc* indica que hi ha hagut un risc al cicle anterior i que, per tant, les instruccions branch necessiten restar 2 al valor del PC que se les propaga. És necessari també un process, perquè es vol que el senyal estigui actiu un cicle després que el risc s'hagi resolt.

```
process (clk, bus_bubbleFetch)   begin
   if (rising_edge(clk)) then
      if (bus_bubbleFetch = '1') then
         bus_branchrisc <= '1';
      else bus_branchrisc <= '0';
      end if;
   end if;
end process;
```

3.2.3 Prova del processador segmentat

Comprovarem el funcionament del processador mitjançant dos programes de prova: El primer inclourà accessos a memòria i una dependència de dades, mentre que, en el segon, es provaran salts i alguna dependència de dades per a veure com es gestionen els riscos de seqüenciament.
El primer programa és el següent:

```
MOVI R0, 64
MOVI R1, 65
MOVI R2, 66
MOVI R3, 67
MOVI R4, 68
MOVI R5, 80
STB 2(R0), R1
STB 3(R3), R2
STB 0(R5), R3
STB -12(R5), R4
STB -2(R0), R5
MOVI R6, 74
LDB R4, -8(R6)
LDB R2, 4(R3)
```

Expressament, el programa només té una dependència de dades que afecti (MOVI R6,74 i el posterior LDB) però, al tenir 6 instruccions d'accés a memòria, es podrà observar la gestió del risc estructural.

A la figura següent veiem l'execució al simulador de les primeres instruccions, on es veu com després de 6 cicles després del començament de interpretació de la instrucció, s'acaba d'executar el MOVI i s'escriuen valors en els registres. En el moment en què es veu la codificació de la instrucció en el cronograma, es troba a l'etapa Fetch. Es veu com, per exemple, 5 cicles després del fetch de la instrucció MOVI R0, 60 (0x5040), el registre R0 té el valor 0x40, és a dir, 60.

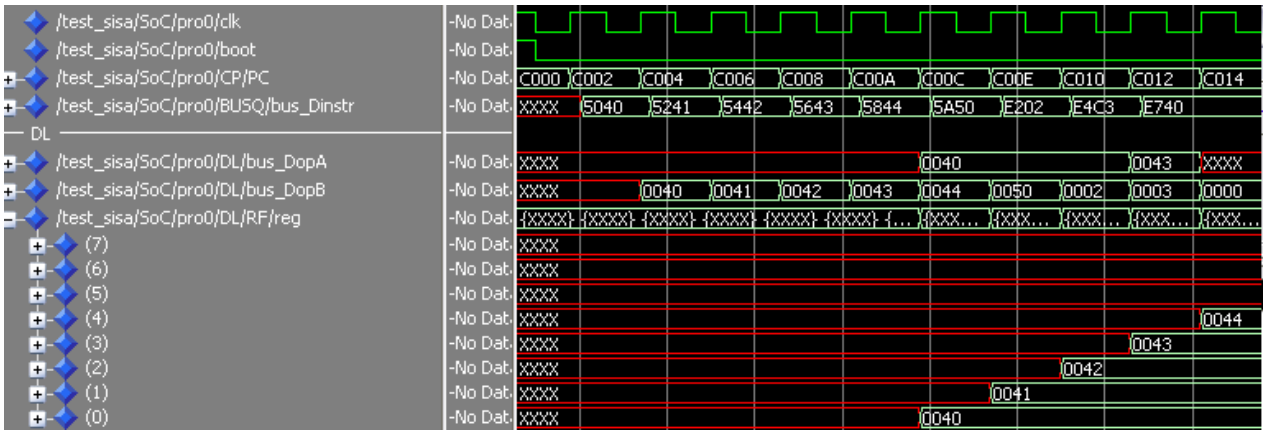


Figura 26: Visió de les 6 etapes

A continuació es mostra l'execució dels 3 primers STB. Quan comença el cronograma, el primer es troba a l'etapa Fetch. Es veu com 3 cicles després, quan està a l'etapa MEM, es bloqueja una instrucció posterior. Un cicle després passa el mateix, doncs el segon STB arriba a l'etapa MEM.

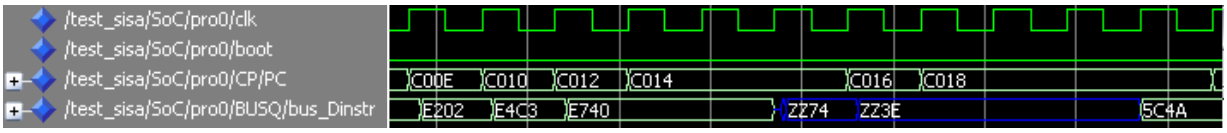


Figura 27: Gestió de riscos estructurals

Per últim, la següent figura mostra l'execució del MOVI i del posterior LD. El cronograma comença amb la instrucció MOVI a l'etapa Fetch. 2 cicles després, quan la instrucció LD està a l'etapa DR, detecta que hi ha una instrucció que fa servir R6 com a operand destí, pel que es bloqueja fins que se solucioni el risc. Es necessita que la instrucció MOVI passi per les etapes ALU, MEM i WB, pel que la instrucció LDB queda bloquejada 3 cicles.

Figura 28: Gestió d'un risc de dades

El segon programa és el següent:

```
movi r0, 0
movi r1, 1
movi r2, 14
movi r3, 3
shl r3, r3, r2
movi r5, 32 ; @ s2
movi r4, 40 ; @ s1
movi r7, 48 ; @ s4
movi r6, 56 ; @ s3
or r4, r4, r3
or r5, r5, r3
or r6, r6, r3
or r7, r7, r3
movi r2, 64
jz r0, r4 ; Salta a s1
halt
```

```
.org 32
s2:
st 2(r2), r1
addi r1, r1, 1
jmp r6 ; Salta a s3
halt
s1:
st 0(r2), r1
addi r1, r1, 1
jnz r1, r5 ; Salta a s2
halt
s4:
st 6(r2), r1
st 8(r2), r3
st 0(r0), r1
halt ; Fi del programa
s3:
st 4(r2), r1
addi r1, r1, 1
jal r3, r7 ; Salta a s4
halt
```

El programa utilitza les primeres instruccions per a crear les adreces de les etiquetes s1, s2, s3 i s4 i poder-les guardar a registres, per a poder-hi saltar amb instruccions jump. A partir d'aquí, el programa executa s1, s2, s3 i s4 en aquest ordre, i no com apareixen escrits en el codi. Per a forçar la comprovació que no s'executa cap instrucció fins que el risc de seqüenciament estigui resolt, es posen instruccions halt després dels salts. Per tant, si algun no s'executa bé, el processador s'atura.

A la figura següent s'observa l'execució de les primeres instruccions, que inclou una dependència de dades que ocasiona risc entre les instruccions MOVI R3,3 i SHL R3, R3, R2. La instrucció que té per PC 0xC012 és el MOVI i, la següent, el SHL. Veiem com la instrucció llegida, 0x090B, manté el valor durant més d'un cicle, ja que s'està retenint.

Figura 29: Detecció del risc de dades

A la següent figura es mostra el salt a s1. Veiem com es busca la instrucció HALT (0xFFFF) però no arriba a executar-se, ja que es queda retinguda fins que la instrucció de salt acaba d'executar-se. Es passa del PC 0xC022 a 0xC02A, que és l'adreça de s1.

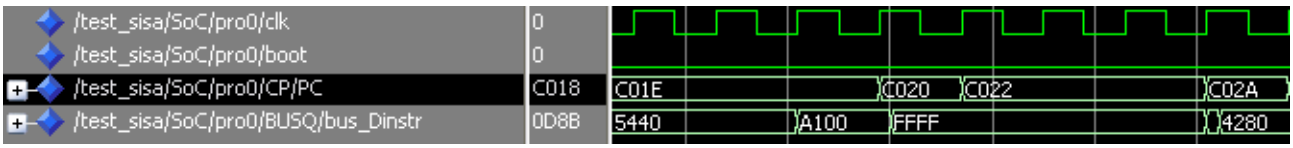


Figura 30: Detecció del primer risc de seqüenciament

A la següent figura es mostra el risc de dades ocasionat per les instruccions ADDI R1, R1, 1 i JNZ R1, R5 i el salt a s2. El JNZ és la instrucció 0xA341 i l'ADDI és la 0x2241. Veiem com el salt queda retingut fins que s'acaba d'interpretar l'ADDI. Després, la instrucció següent al salt, el HALT, queda retinguda fins que s'acaba d'interpretar el salt, moment en què es trenca el seqüenciament i se salta a l'adreça 0xC022.

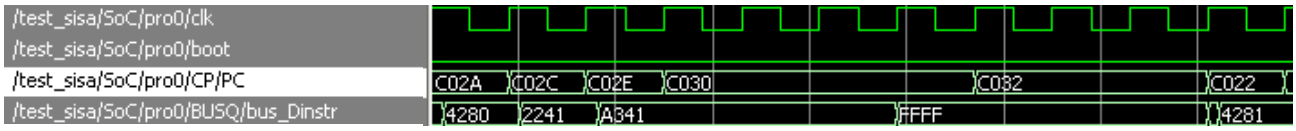


Figura 31: Detecció del segon risc de seqüenciament

Per últim, es mostra el salt a s4 i com es bloqueja el tercer dels Store quan el primer i el segon arriben a l'etapa MEM. La instrucció amb PC 0xC042 és la següent al salt, que queda retinguda fins que aquest s'acaba d'interpretar i se salta a l'adreça 0xC032. En aquesta hi ha un ST, pel que 3 cicles després de la seva descodificació (quan el PC val 0xC036 al cronograma) es troba a l'etapa de memòria, pel que reté la instrucció que s'està buscant. Això passa durant dos cicles perquè la segona instrucció també accedeix a memòria.

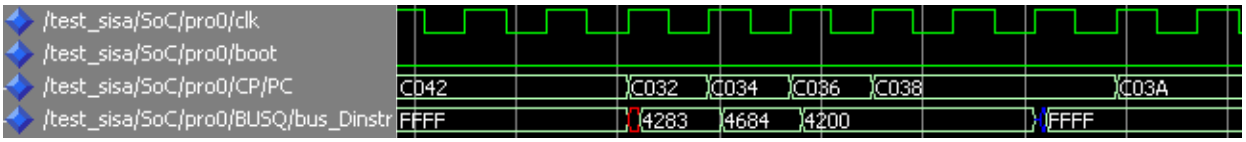


Figura 32: Últim salt del programa de prova i accés a memòria

3.3 Reducció de la latència

3.3.1 Curtcircuits

Per a implementar els curtcircuits, cal modificar la lògica del controlador del pipeline i afegir els multiplexors a l'etapa DR que els permetin transferir dades.

Pel que fa al controlador del pipeline, s'afegeixen sis senyals noves, *bypassExea*, *bypassExeb* *bypassMema*, *bypassMemb*, *bypassWBa* i *bypassWBb*, que indiquen de quina etapa prové el curtcircuit i a quin operand (a o b) s'apliquen. Per l'operand a, aquestes senyals valen '1' si l'adreça del registre font coincideix amb l'adreça del registre destí que usa la instrucció vella i aquesta és una instrucció que escriu a registres. Per l'operand b, cal comprovar el mateix i, a més, que la instrucció jove utilitzi el registre b. Aquesta informació l'aporta el senyal *Rb_immed*, que es fa servir a l'etapa DR i, per tant, no cal que sigui propagat. Cal notar que el curtcircuit a l'etapa WB és necessari perquè no es pot escriure i llegir en un registre en el mateix cicle i que el valor llegit correspongui al valor escrit.

```
bypassExea <= '1' when (addr_a_Dec = addr_d_Exe and destIsUsedExe = '1' ) else
              '0';

bypassExeb <= '1' when (addr_a_Dec = addr_d_Exe and destIsUsedExe = '1' and
                        Rb_immed = '1') else
              '0';
```

Amb l'addició dels curtcircuits, només pot donar un risc de dades en el cas en què una instrucció utilitzi com a registre font el valor produït per la instrucció anterior i aquesta sigui un Load. Això és degut a que la distància entre l'etapa productora (MEM) i la consumidora (DR) és de 2.

Per tant, cal modificar els senyals que introdueixen bombolles al pipeline: només cal introduir-les per riscos de dades en el cas anterior, el que es tradueix a:

```
bus_bubblePC <= '1' when (addr_a_Dec = addr_d_Mem or addr_b_Dec = addr_d_Mem) and
                         (destIsUsedMem = '1' and load = '1') else
                '0';
```

On el senyal *load* és un senyal obtingut a l'etapa DR i que es propaga fins l'etapa MEM, i indica si la instrucció és un LD o un LDB.

D'altra banda, pel que fa a l'etapa DR, cal afegir un multiplexor per cada senyal nou que genera el

controlador del pipeline. Per a aconseguir, en cas que més d'una etapa pugui fer el curtcircuit, obtenir la dada més nova, es fan servir 3 multiplexors 2 a 1 per operand. El primer dels multiplexors, controlat pel senyal *bypassWB*, té com a entrades el valor del banc de registres i la dada curtcircuitada de l'etapa WB. La sortida d'aquest multiplexor alimenta l'entrada d'un altre, controlat pel senyal *bypassMem* i que té com a segona entrada la dada curtcircuitada de l'etapa MEM. Per últim, la sortida d'aquest multiplexor alimenta el tercer, controlat pel senyal *bypassExe*, i que té com a segona entrada la dada curtcircuitada de l'etapa ALU. Aquest últim multiplexor és el que es connecta amb el registre de desacoblament de l'operand corresponent. El codi que ho implementa és el següent:

```
with bypassWBa select
   bus_0Mem <= bus_opaRegfile when '0',
               data_bypassWB when others;

with bypassMema select
   bus_0Exe <= bus_0Mem when '0',
               data_bypassMem when others;

with bypassExea select
   bus_Dopa <= bus_0Exe when '0',
               data_bypassExe when others;
```

3.3.2 Predictor estàtic de salt

El predictor de salt ha de donar la informació de si un salt és prediu o no, a quina adreça se salta i de si s'han d'avortar instruccions per culpa d'una predicció mal feta. El SISA-3 té dos tipus d'instruccions de salt: els jump salten a l'adreça que indica el contingut d'un registre font, mentre que els branch sumen un desplaçament al valor del PC. Com que el predictor s'usa a l'etapa DR mentre que la direcció efectiva del branch se sabia abans a l'etapa ALU, es necessitarà afegir un sumador. Aquesta lògica, equivalent a l'operació CMPS a l'ALU, és necessària per a efectuar la predicció i per a indicar l'adreça on s'ha de saltar en cas que es predigui que es trenca el seqüenciament.

Per a generar els senyals de sortida del predictor, cal: informació del tipus de salt, el PC, el registre que indica l'adreça del salt en els JMP i l'immediat a sumar en els branch. Si s'està interpretant un branch, es mira si l'immediat és negatiu. En cas afirmatiu, com els branch salten relativament al PC, se sap que és un salt enrere i es prediu que se saltarà. Si l'immediat és positiu, es prediu que no se

salta. Si s'està interpretant un JMP, es mira si el contingut del registre que indica l'adreça del salt és menor a PC+2, que correspon a l'adreça de la següent instrucció en seqüència. En cas que sigui menor, es prediu que se salta i, en cas contrari, es prediu que no.

El senyal que indica la predicció es guarda durant un cicle, moment en què la instrucció es troba a l'etapa ALU i s'avalua la condició de salt. Aquesta avaluació s'envia al predictor, i la fa servir junt amb el senyal de predicció per a generar un nou senyal que indica si la predicció ha estat errònia. Si s'havia predit seguir en seqüència, la següent instrucció es trobarà a l'etapa DR. Si s'havia predit no seguir en seqüència, la següent instrucció es trobarà a l'etapa PC, i s'haurà injectat una NOP a l'etapa F de la instrucció següent al salt, per a anul·lar-la. El senyal de predicció incorrecta es fa servir junt amb el de predicció per a produir una bombolla a l'etapa PC, injectar una NOP a l'etapa F i injectar una NOP descodificada a l'etapa DR en el cas d'haver seguit el seqüenciament i per a produir una bombolla a l'etapa PC i injectar una NOP a l'etapa F en cas d'haver trencat el seqüenciament.

El codi que implementa aquesta lògica es mostra a continuació. Per una banda, els senyals del predictor:

```
predict <= '1' when (branch = '1' and immed < x"00") or
                    (jmp = '1' and address < PC+2) else
           '0';

predictedAddr <= PC+2+(immed(7)&immed(7)&immed(7)&immed(7)&immed(7)
                 &immed(7)&immed(7)&immed(7 downto 0)&'0') when branch = '1' else
                 address when jmp = '1';

process(clk, predict)
   begin
      if (rising_edge(clk)) then
         last_prediction <= predict;
      end if;
end process;

misprediction <= '1' when (last_prediction = '1' and ev = '0') or
                          (last_prediction = '0' and ev = '1') else
                 '0';
```

D'altra banda, cal canviar els senyals de bombolla:

```
bus_bubblePC <= '1' when ((addr_a_Dec = addr_d_Mem or addr_b_Dec = addr_d_Mem) and
                         (destIsUsedMem = '1' and load = '1') or
                          misprediction = '1' else
                '0';

bus_bubbleFetch <= '1' when ((addr_a_Dec = addr_d_Mem or addr_b_Dec = addr_d_Mem) and
                         (destIsUsedMem = '1' and load = '1') or
                         (misprediction = '1' and last_predicion = '0')) else
```

I, per últim, cal actualitzar la lògica de l'etapa PC per a que, si es prediu trencar el seqüenciament, el següent valor que prengui el PC sigui l'adreça predita:

```
bus_DPC <= MuxOut  when bubblePC = '0' or (bubblePC = '1' and tknbr = "10") else
           PC when bubblePC = '1' and tknbr /= "10" else
           predictedAddr when predict = '1';
```

3.4 Entorn de demostració

No és fàcil trobar una manera per a ensenyar què està fent el processador. A més, a simple vista, sembla que no faci res de diferent respecte el projecte original. Per això, aquest treball proposa afegir un mode de funcionament anomenat "Mode Demo". Només es llegeixen 8 dels 10 interruptors de la placa al fer una interrupció o una instrucció IN. Com el SW[8] està reservat per a que sigui el senyal *boot*, s'usarà el SW[9] per a activar o desactivar el Mode Demo.

En aquest mode, es desactiven les interrupcions, i s'ignoren les instruccions IN i OUT. A més, el temps de cicle augmenta fins a ser aproximadament 1.5 segons.

Al mòdul de més alt nivell (el que té contacte amb el rellotge de la placa), s'implementa un comptador de 24 bits que, a cada flanc ascendent del rellotge de 50 MHz de la placa, augmenta en 1. El rellotge que se li transmet al processador no és el de 50 MHz, sinó un dels bits d'aquest comptador. Com aquest bit, gràcies a la suma, canvia de valor a intervals regulars, pot funcionar de rellotge pel processador. Depenent de quin bit s'agafi la divisió de freqüència serà més gran. Per exemple, si s'agafa el bit 0, el processador funcionarà a 25 MHz i, si s'agafa el bit 3, a 3,125 MHz. Degut al temps que necessita tant la memòria com el divisor, el processador funciona a 12.5 MHz normalment.

El SW[9] fa de senyal de control d'un multiplexor que selecciona entre el bit 1 i el bit 24 del comptador per a fer de rellotge. Si el SW[9] no està actiu, el rellotge del processador és de 12.5MHz mentre que si ho està, el rellotge del processador és d'1.49 Hz.

Aquest interruptor també fa de senyal de control d'un multiplexor que selecciona entre el senyal que controla el controlador d'entrada/sortida amb els LED vermell i verds i dos nous senyals que surten del controlador del pipeline. Quan el SW[9] no està actiu, es treballa en el mode normal del processador, en què els LED prenen per valor el que diu el controlador d'entrada/sortida. Quan es treballa en Mode Demo, se seleccionen els senyals del controlador del pipeline.

Els LED vermells [5:0] representen les 6 etapes del pipeline, sent el LEDR[5] l'etapa CP i el LEDR[0] l'etapa WB. Quan una etapa ocasioni un risc, el LED corresponent s'encendrà. A més, els tres LED de més pes s'encendran indicant el tipus de risc: LEDR[7] serà un risc de dades, LEDR[8] salt mal predit i, LEDR[9], un risc estructural.

Els LED verds[5:0] representen les etapes del pipeline de la mateixa forma que els vermells. Quan es produeixi un curtcircuit, els LED corresponents a l'etapa productora i a l'etapa consumidora s'encendran.

A la primera de les següents imatges es veu la detecció d'un risc de dades per culpa de la seqüència d'instruccions LD R5, 0(R0), ADD R5,R5,R5 i una qualsevol. A l'arribar a l'etapa ALU (LEDR[2]), la instrucció LD causa un risc de dades (LEDR[7]) perquè encara no té el valor disponible. A la segona de les imatges, es veu la mateixa seqüència un cicle més endavant: el risc de dades ha desaparegut gràcies al curtcircuit de l'etapa MEM a l'etapa DR, però l'accés a memòria de dades provoca un risc estructural (LEDR[9]), que bloqueja durant un cicle la tercera instrucció. El valor carregat al registre R5 és enviat a l'etapa DR a través del curtcircuit MEM->DR (LEDG[1] i LEDG[3]).

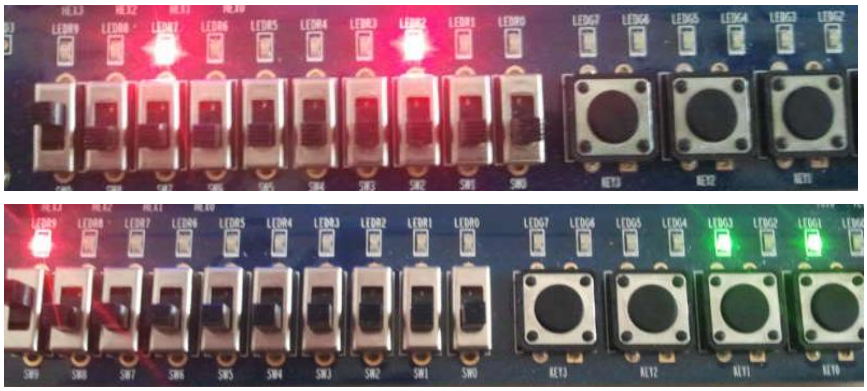


Figura 33: Mode Demo del processador

4. Resultats

En aquest capítol, es proposa comparar el rendiment del processador implementat amb el processador original mitjançant l'execució de 3 programes. Tot i que l'objectiu del projecte no era millorar el rendiment del processador, s'ha cregut convenient mesurar numèricament què s'ha obtingut i com afectarien les següents millores.

4.1 Processadors comparats

L'anàlisi s'efectua en 5 diferents processadors, dels quals 2 han estat implementats. Dels altres 3, un s'usa com a límit inferior dels cicles que l'execució d'un programa podria trigar i la resta són millores que no s'han arribat a implementar, però de les que s'avalua el rendiment que podrien aportar al processador implementat en aquest projecte.

Processadors implementats:

Processador original: Cada instrucció triga 2 cicles a ser executada.

SegSISA: El processador implementat per aquest projecte.

Processadors teòrics per a fer comparacions:

Segmentat Pur: El processador original amb dos busos per a accedir a memòria. D'aquesta manera, es pot segmentar en dues etapes i començar l'execució d'una nova instrucció a cada cicle.

SegSISA-2: El SegSISA amb dos busos per accedir a memòria. D'aquesta manera, desapareix el risc estructural.

SegSISA-TC: El que limita la freqüència del processador és la implementació actual del multiplicador i el divisor. En aquesta versió del SegSISA, s'augmenta la profunditat del pipeline a 7 etapes, estant l'etapa d'execució dividida en dues. Les instruccions de multiplicació i divisió trigarien 7 cicles en ser executades, mentre que totes les altres trigarien 6. La segmentació de l'execució d'aquestes operacions aritmètiques permet reduir el temps de cicle a la meitat.

4.2 Programes de prova

S'han escrit tres programes per observar el rendiment del processador en diferents situacions i com aquest varia depenent de les millores que s'hi apliquen. Un dels programes efectuarà només càlculs aritmètics, un altre efectuarà un número elevat d'accessos a memòria i, el tercer, inclourà

instruccions que provocaran que el predictor de salt no encerti sovint.

Càlcul de l'ièssim número de Fibonacci

```
       IN R0; R0 número de la sèrie que es vol calcular
       MOVI R1, 0
       MOVI R2, 1
       MOVI R3, 2
Bucle: CMPLE R5, R3, R0
       BZ R5, Fibucle
       ADD R4, R1, R2
       ADDI R1, R2, 0
       ADDI R2, R4, 0
       JMP Bucle
Fibucle: HALT
```

Aquest programa executa el següent codi:

```
  int u = 0;
  int v = 1;
  int i, t;

  for(i = 2; i <= n; i++){
    t = u + v;
    u = v;
    v = t;
  }
  return v;
```

Producte de vectors

```
       MOVI R0, 0; @A
       MOVI R1, 100; @B
       MOVI R2, 200; @C
       MOVI R3, 100; N
       MOVI R4, 0; i
Bucle: CMPLT R5, R4, R3
       BZ R5, Fibucle
       LD R6, 0(R0)
       LD R7, 0(R0)
       MUL R7, R6, R7
       ST 0(R2), R7
       ADDI R0, R0,1
       ADDI R1, R1, 1
       ADDI R2, R2, 1
       ADDI R4, R4, 1
       JMP Bucle
Fibucle: HALT
```

Aquest programa executa un bucle `for(i=0; i < N; ++i) C[i] = A[i]*B[i].`

Programa amb salts variables

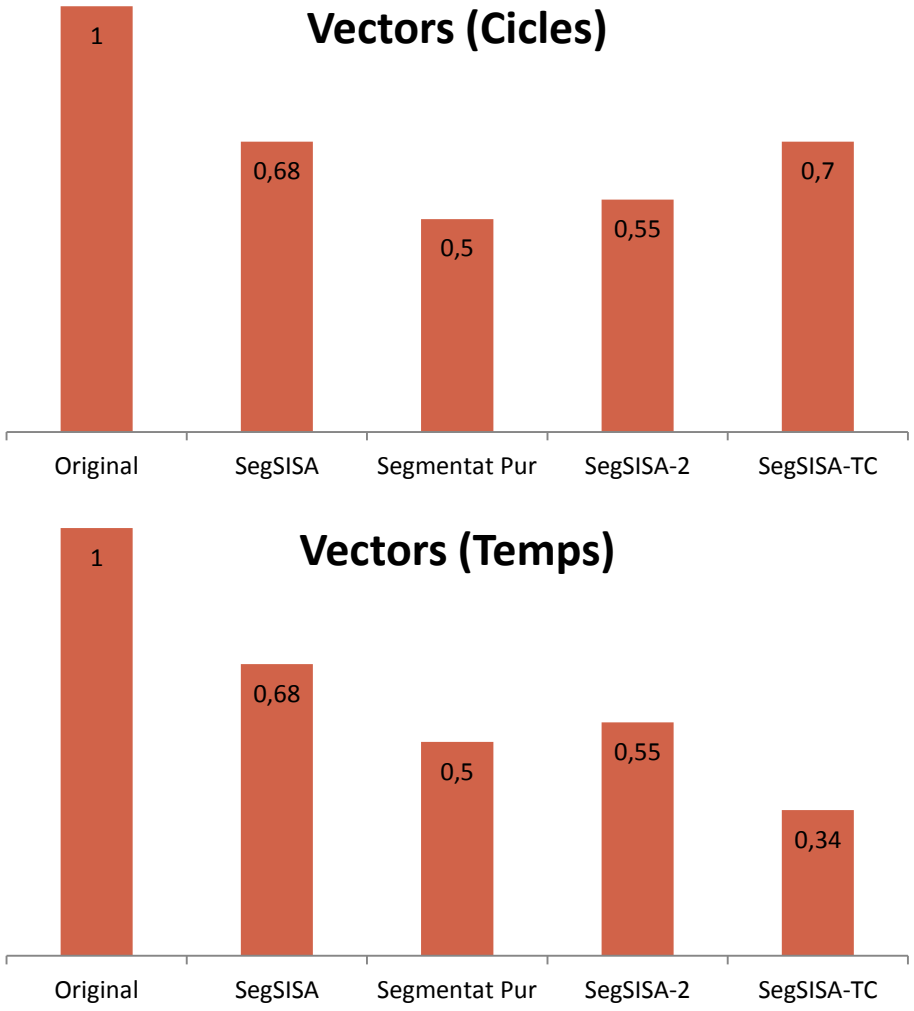
```
        MOVI R0, 0
        MOVI R1, 255
        MOVI R3, 3
Bucle:  ADDI R0, R0, 1
        CMPLE R2, R0, R1
        BZ R2, Fibucle
        AND R4, R0, R3
        BNZ R4, Bucle
        ADDI R5, R5, 1
        JMP Bucle
Fibucle: Halt
```

Aquest programa executa el següent codi:

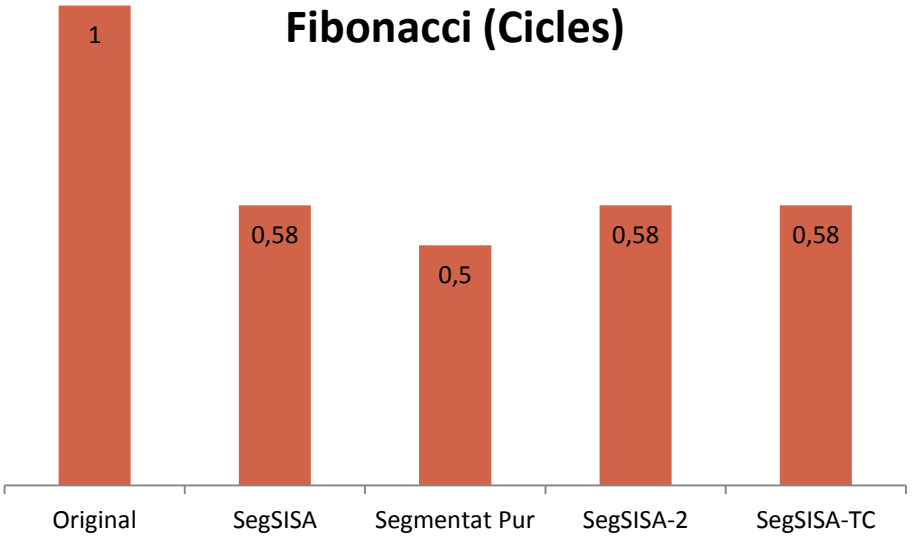
```
while (i < N){
     if(!(i&3)) ++variable;
     ++i;
}
```

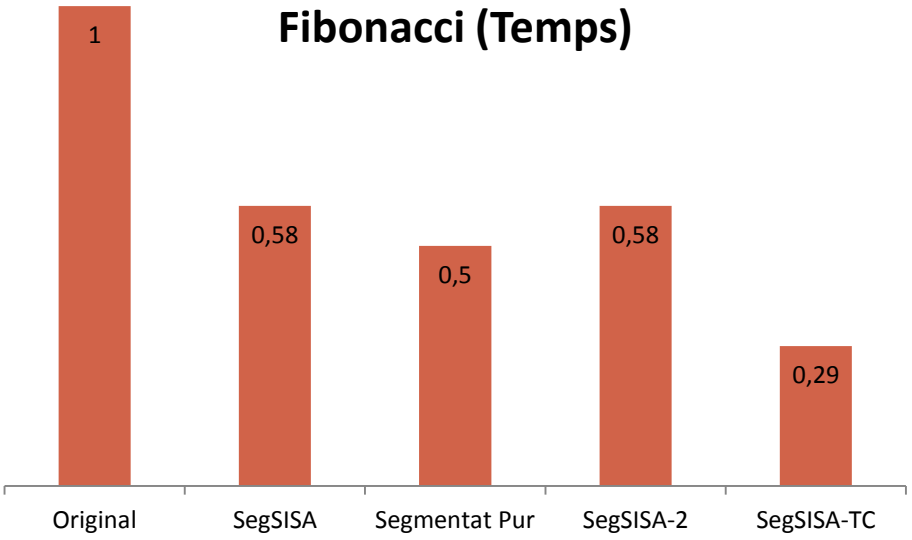
4.3 Comparacions

A les gràfiques següents es comparen els 5 processadors. Totes les gràfiques estan normalitzades. A la barra de més a l'esquerra del gràfic està el processador original com a referència del temps i número de cicles màxim dels programes. El bucle dels programes s'executa prou iteracions com per a que els cicles corresponents a les instruccions d'inicialització i els corresponents a les instruccions finals siguin negligibles.

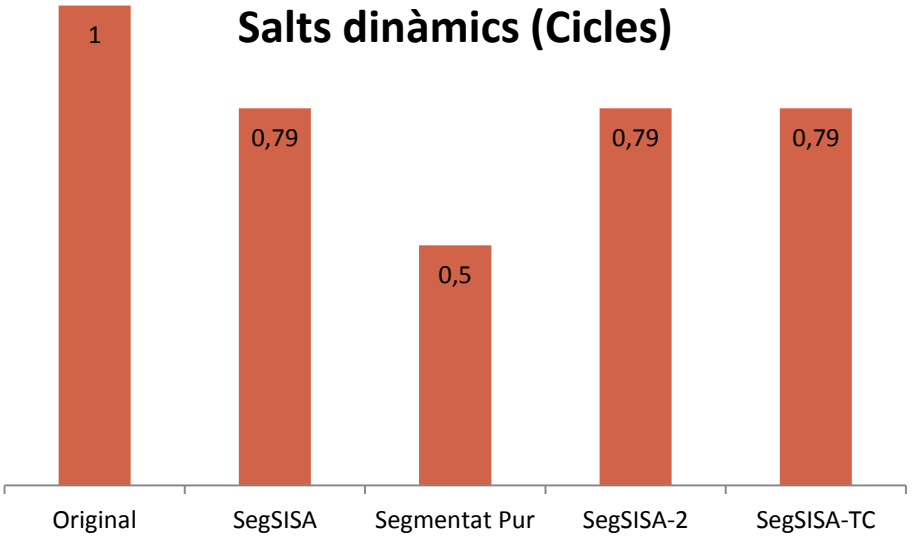


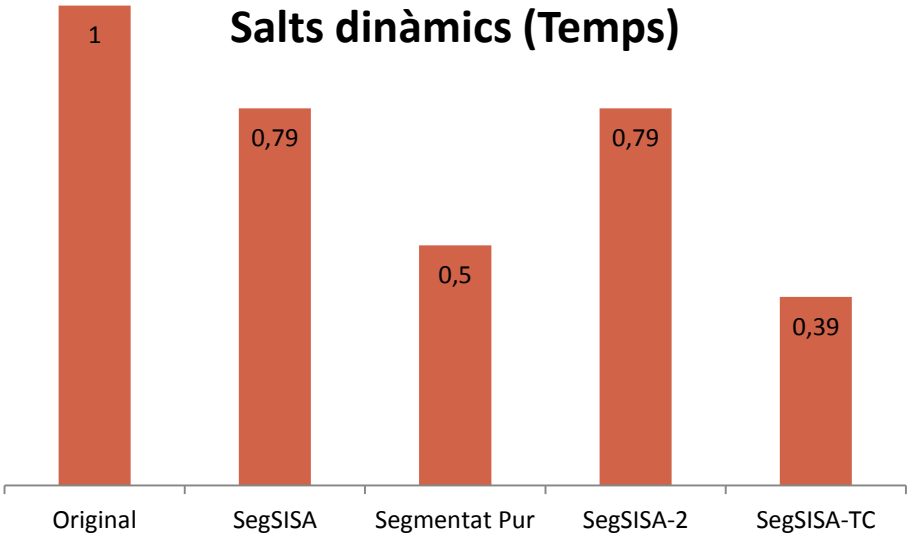
Els cicles que triga el segmentat pur són el mínim número de cicles possibles. Com que el programa té una alta càrrega d'accessos a memòria, el risc estructural perjudica el rendiment del SegSISA. A més, com que una de les instruccions és una multiplicació, el TC/2 perd un cicle més per iteració. Veiem que el que realment afecta al rendiment és l'accés a memòria, doncs si s'afegeixen dos busos, el processador tindria un CPI de gairebé 1: el SegSISA perd 4 cicles per iteració (3 de risc estructural i 1 pel salt) i, el SegSISA-2, només en perdria 1. Tot i perdre 1 cicle més per iteració per culpa de la multiplicació, la reducció del temps de cicle aconsegueix que el SegSISA-TC funcioni un 45% més ràpid que el segmentat pur.





En aquest exemple, no hi ha riscos estructurals i els de dades queden emmascarats pels curtcircuits. Es perd un cicle per iteració per culpa del risc de seqüenciament, ja que és el cost de predir saltar enrere i encertar. A més, com no es fan operacions de multiplicació ni de divisió, el SegSISA-TC també triga el mateix número de cicles, encara que aquests van el doble de ràpid que en els altres processadors i, per tant, triga gairebé 1/4 del temps del processador original.





El bucle, en el SegSISA, triga 9 cicles en executar-se si el predictor encerta l'if, mentre que en triga 11 si no l'encerta. El codi força que el predictor falli el 75% de les vegades que executa la instrucció condicional. Com tots els salts són enrere, el predictor sempre indicarà que cal trencar el seqüenciament, però el codi només el trenca el 25% de les iteracions. Per tant, en mitjana, es triga gairebé un 60% més de cicles que el processador segmentat pur a executar el bucle. No hi ha risc estructural i els riscos de dades queden emmascarats gràcies als curtcircuits. Per aquesta raó, les 3 versions del SegSISA triguen el mateix número de cicles: cap d'elles canvia el predictor. El segmentat pur no pateix riscos de seqüenciament, pel que aconsegueix anar el doble de ràpid que l'original. Veiem, una altra vegada, com l'augment de la freqüència és millor que l'eliminació dels riscos: la versió amb 7 etapes va un 40% més ràpid que el processador segmentat pur.

5. Anàlisi temporal i econòmic

En aquest capítol es fa una breu explicació de com s'ha gestionat el temps per a desenvolupar el projecte, així com el cost dels recursos hardware i humans emprats per a acabar-lo satisfactòriament.

5.1 Anàlisi temporal

El projecte comença a principis de febrer de 2013 i preveu la seva finalització a mitjans de juny del 2013. Es divideix en tres tasques: implementació del mode sistema, implementació del pipeline i gestió dels riscos. En totes tres, per a treballar, se segueixen els següents passos:

- Anàlisi de les noves funcionalitats i estudi de com incorporar-les. S'avalua l'estat del processador (quins mòduls el composen i com es comuniquen) així com els mòduls nous a afegir. Es realitza un disseny en forma d'esquema de blocs on es visualitzen els nous components i tots els senyals que inclouen o que és necessari afegir o modificar.

- Reunió prèvia a començar a escriure codi amb el director per a valorar les solucions proposades. S'usa aquesta reunió amb finalitats de seguiment i de validació: per una banda el director comprova que s'ha començat la següent etapa del projecte i, per altra, s'assegura que el disseny és coherent i soluciona els problemes proposats.

- Escriptura del codi. Elaborar codi (sobretot en VHDL) és complex, pel que s'usaran eines de simulació de circuits i Netlist viewers per a comprovar que el codi compleix el disseny intencionat i que cada senyal que composa el codi fa el que es proposava.

- Creació d'uns jocs de proves exhaustius. Aquest és un pas crític, degut a que la correctesa del programat serà determinada per la superació d'aquests. Per tant, cal comprovar dues coses. Primerament, tot i que el nombre d'entrades és infinit, existeix un subconjunt finit tal que afirma que l'etapa compleix l'especificació. Segon, la forma de les entrades d'aquest conjunt és la que s'espera en totes les execucions. Assegurant que els jocs de proves compleixen aquestes condicions, s'assegura també que la seva superació implica el bon funcionament del codi.

- Comprovació en el simulador i en la placa del funcionament de les modificacions. El mètode de validació és l'execució del codi amb els jocs de proves prèviament dissenyats com a entrada. Primer es començarà executant-lo al simulador, ja que és l'eina que permet veure de forma més ràpida i senzilla el comportament del circuit, a més de tenir una gran capacitat de depuració. Una vegada funcioni al simulador, es prosseguirà a l'execució a la placa. La superació dels jocs de proves en aquesta implica la finalització de l'etapa, a l'acomplir-se l'objectiu d'executar codi funcional a la FPGA. Si no se superen els jocs a la placa, caldrà utilitzar un analitzador lògic per a veure en quin moment de l'execució apareixen els problemes.

- Reunió amb el director per a comentar els resultats i/o problemes obtinguts. Aquest és el segon mètode de seguiment. Mitjançant aquesta trobada, que pot ser replicada a un moment previ on hi hagi proves que funcionen correctament i altres que no, el director sap quin és l'estat del processador, què s'està provant i de quines maneres, i pot aconsellar afegir casos de prova o aportar visions de per què algunes poden fallar.

Aquesta metodologia encaixa en el calendari com mostra el diagrama de Gantt de la figura 34.

El projecte ha estat desenvolupat en, aproximadament, 480 hores, distribuïdes entre 17 setmanes.

5.2 Anàlisi econòmic

Recursos humans

Al llarg del projecte hi ha dos rols clarament diferenciats: el d'analista hardware i el de programador. La complexitat implícita a l'anàlisi i disseny és superior a la de la posterior programació, i això s'ha de tenir en compte. Tanmateix, les hores de programació inclouen les de depuració i proves, que són gran part del temps dedicat al projecte.

	Hores	Preu/hora	Total
Analista hardware	35	25€	875€
Programador	445	15€	6675€
Total	480		**7750€**

Aquestes hores corresponen a les que es veuen al diagrama de Gantt. S'ha tingut en compte que és l'analista qui s'encarrega d'assistir a les reunions, ja que s'assumeix que aquestes comporten la verificació de la correctesa del disseny abans d'escriure el codi.

Recursos hardware

Es considera que la potència de les versions de no pagament del programari d'Altera és suficient per a dur a terme el projecte. La placa de desenvolupament en què s'implementa el processador és la DE1 d'Altera, com s'ha vist i justificat anteriorment.

A la següent taula es mostren els costos detallats:

	Preu
Eines per a compilar (software amb GPL)	0€
Quartus II (Web edition)	0€
ModelSim (Starter edition)	0€
Ordinador personal i Sistema Operatiu	570€
Placa Altera DE1	200€
Total	**770€**

	Nombre	Inicio	Fin	Predecesoras
1	**MODE SISTEMA**	**12/02/2013**	**21/02/2013**	
2	Estudi, anàlisi i jocs de proves	12/02/2013	12/02/2013	
3	Interrupcions	15/02/2013	19/02/2013	2
4	Simulacions i solució de problemes	20/02/2013	21/02/2013	3
5	**PIPELINE**	**21/02/2013**	**21/05/2013**	
6	Estudi, anàlisi i jocs de proves	21/02/2013	21/02/2013	
7	Reunió prèvia a l'escriptura del codi	22/02/2013	22/02/2013	6
8	Modificacions al datapath	25/02/2013	08/03/2013	7
9	Modificacions a la lògica de control	11/03/2013	22/03/2013	7
10	Gestió de riscos estructurals	25/03/2013	29/03/2013	8,9
11	Gestió de riscos de dades	01/04/2013	16/04/2013	8,9
12	Gestió de riscos de seqüenciament	17/04/2013	26/04/2013	8,9
13	Simulacions i solució de problemes	29/04/2013	21/05/2013	10,11,12
14	**REDUCCIÓ DE LA LATÈNCIA**	**21/05/2013**	**10/06/2013**	**5**
15	Estudi, anàlisi i jocs de proves	21/05/2013	22/05/2013	
16	Reunió prèvia a l'escriptura del codi	22/05/2013	22/05/2013	15
17	Implementació dels curtcircuits	23/05/2013	27/05/2013	16
18	Predictor de salt	27/05/2013	29/05/2013	16
19	Simulacions i solució de problemes finals	30/05/2013	10/06/2013	17,18
20	Elaboració de la memòria	22/05/2013	13/06/2013	5
21	Preparació de la lectura	14/06/2013	20/06/2013	20

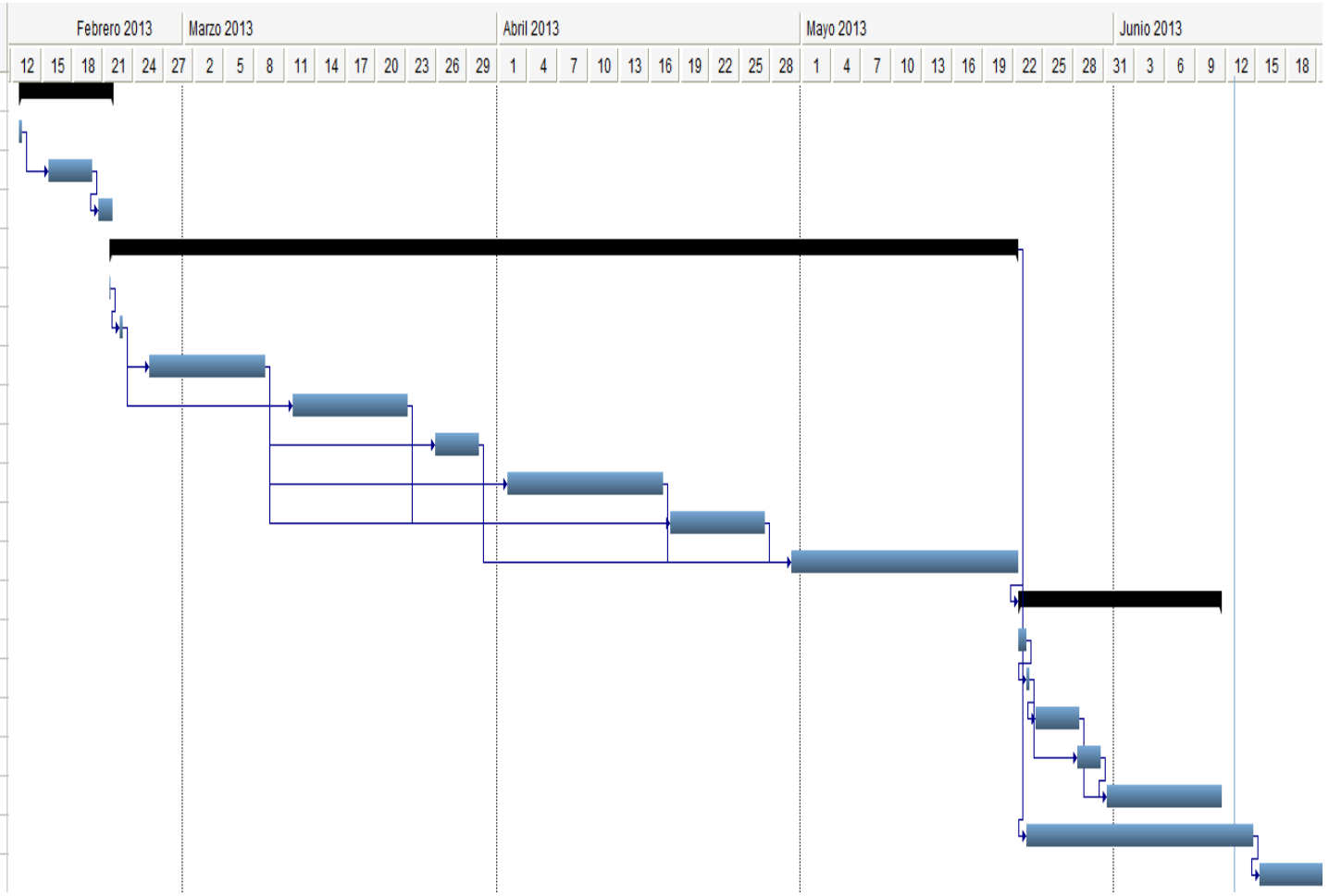


Figura 34: Diagrama de Gantt del projecte

6. Impacte social

En aquest capítol, es tracten els aspectes ambientals i legals del projecte. Es parlarà de la justificació de la sostenibilitat del desenvolupament del treball i, a continuació, s'explicaran les diferents normatives i regulacions que afecten al projecte.

6.1 Desenvolupament sostenible

Es pot mesurar la contribució al desenvolupament sostenible de la tecnologia mitjançant dos indicadors, que es refereixen a la relació d'una tecnologia amb el sistema natural [8]:

1- Indicadors d'eco-eficiència, que mostren l'impacte de la desmaterialització dels recursos.
2- Indicadors de qualitat ambiental, que mostren l'impacte sobre el medi ambient generat per la tecnologia.

És clar que aquest projecte és eco-eficaç, ja que la FPGA, el recurs principal utilitzat, és altament renovable. Això és degut a la seva capacitat contínua de programació, que permet aprofitar-lo en projectes diferents. D'altra banda, aquest component també compleix satisfactòriament el segon tipus d'indicador , ja que el seu ús no emet nocivament components nocius a l'aigua, terra o atmosfera.

Una eina molt usada per a estudiar aquests indicadors és l'ACV, l'anàlisi del cicle de vida. L'ACV analitza l'impacte mediambiental d'un disseny tenint en compte totes les fases de vida del producte (adquisició de matèries primes, processat i fabricació, distribució, ús i reutilització i final del cicle de vida- residus generats i reciclatge) i calcula la contribució del producte en totes elles a les diferents formes de contribució.

Tot i que aquest estudi no pot valorar la gran part d'aquestes fases, és important recalcar que l'ACV no suma els impactes, sinó que els pondera. És a dir, són més transcendents les etapes finals de la vida del producte (on es reutilitza i genera residus per la seva destrucció) que no el que el gas que s'emet, per exemple, al transportar-lo [9]. Per tant, l'alta reusabilitat de les FPGA és el factor clau que fa que el projecte sigui sostenible.

6.2 Regulacions

Tal i com està plantejat el projecte (aprofitar els resultats pràctics d'una assignatura i combinar-los amb coneixements d'un altre per a fer quelcom més gran amb una finalitat purament didàctica), no hi ha cap entrebanc legal que n'impedeixi la realització o el progrés. És interessant, però, valorar què s'hauria de tenir en compte en el cas que el resultat del projecte es volgués comercialitzar.

En primera instància, apareix el fet de construir un producte que s'ha de voler vendre, i cal mirar què s'aprofita d'altres i si realment es pot fer servir. La gran majoria de codi s'escriu des de zero, i només es reutilitza codi de llibreries. Aquestes llibreries (std_logic i std_numeric) poden ser usades sense problema [10][11].

Una vegada el producte estigués construït, el següent pas és veure les limitacions d'exportació i consum. Tant la placa [12] com el Cyclone II [13] estan dins dels estàndards europeus (EC) i americans (FCC) sobre compatibilitat electromagnètica.

Per últim, com el projecte es realitza en una FPGA (component altament reutilitzable), compliria la directiva sobre residus d'aparells electrònics i elèctrics (WEEE) [14].

7. Conclusions

En aquest capítol final es comentaran les conclusions que s'extreuen de la finalització del projecte, comparant els objectius inicials amb els resultats, el que s'ha hagut de treballar per a aconseguir el que es volia i les possibles línies de futur que es proposen pel processador.

S'han aconseguit complir satisfactòriament els objectius plantejats inicialment: modificar el processador de l'assignatura PEC implementant el mode sistema juntament amb les interrupcions i implementant la segmentació i la gestió dels riscos segons les guies teòriques explicades a l'assignatura AC2. El processador executa la gran majoria d'instruccions del repertori definit, exceptuant les que tracten amb coma flotant i el TLB. Els conceptes que s'havien d'assimilar per a la realització del projecte s'han aplicat correctament, obtenint un processador segmentat funcional a la placa DE1 d'Altera. Tot i que hi ha certes parts que s'han implementat amb aquest dispositiu en ment, es podrien adaptar fàcilment a un altre.

Durant el projecte ha estat necessari millorar el domini dels simuladors i analitzadors lògics, així com del coneixement del llenguatge VHDL per a arribar a complir els objectius. S'ha estudiat l'arquitectura SISA-F i SISA-3, i s'han vist els canvis en el seu disseny per a poder-los dur a terme. Finalment, s'ha aprofundit en els coneixements sobre segmentació, detecció de riscos i curtcircuits, a l'haver fet funcionar en un dispositiu físic tota una sèrie d'idees conceptuals.

Els resultats obtinguts són similars al que s'esperava, al no reduir la freqüència a la que funciona el processador. Tot i que no es buscava millora, el fruit del projecte és productiu en programes que no fan un gran ús de la memòria, ja que aquesta provoca un risc no mitigable amb les tècniques que s'usen per a tractar altres riscos.

Les possibilitats d'ampliació del projecte són grans, tenint en compte la diferència amb un processador real funcional en una FPGA.

D'entrada, s'ha limitat el projecte a utilitzar només la SRAM per a il·lustrar el disseny de la solució d'un risc addicional als tractats a l'assignatura AC2. Això és inviable en una arquitectura real, pel que es podria incorporar un controlador per la memòria Flash. Aquesta solució separaria les

memòries en memòria de dades (SRAM) i memòria d'instruccions (Flash), el que eliminaria el risc estructural i no és complicada d'implementar, ja que no s'escriu a memòria d'instruccions.

A continuació, es pot completar el repertori d'instruccions del SISA-3. Per a aconseguir-ho, caldria implementar el TLB i la lògica per a interpretar les instruccions corresponents, així com els circuits necessaris per a operar en els half-precision floats del SISA-3.

Per a seguir amb el primer objectiu del projecte (l'apropament a un processador real mitjançant la implementació de les interrupcions), la següent etapa podria correspondre a la implementació de d'un nucli de sistema operatiu que gestioni les excepcions i les crides a sistema per part de les aplicacions d'usuari.

Finalment, per a tractar millores respecte la segmentació, es podrien aplicar diverses solucions. Per exemple, dividir en diverses subetapes l'etapa d'execució de tal forma que les instruccions de divisió les utilitzin totes mentre que les que no fan servir el divisor només passin per una de les etapes. Això permetria reduir el temps de cicle del processador. Una segona millora podria consistir en canviar el predictor. L'actual s'ha implementat perquè era el que s'ensenya a l'assignatura AC2. Tot i funcionar molt bé en bucles, pot no tenir un rendiment bo en sentències condicionals del tipus if-then-else si aquestes trenquen sovint el seqüenciament. L'ús d'un predictor que guardi una història de salts anteriors (com un predictor de salts local, o un predictor amb comptador saturat) milloraria la interpretació d'aquest tipus d'instruccions sense perjudicar el bon resultat que ja tenen els bucles.

8. Bibliografia

[1] Kaur, Gaganpreet. *VHDL Basics to Programming*. Capítol 2.

[2] Kafig, William. *VHDL 101: Everything you need to know to get started.* Capítol 3.

[3] *STORM SoC*. (Febrer de 2012). http://opencores.org/project,storm_soc

[4] *MIPSr2000*. (Octubre de 2012). http://opencores.org/project,mipsr2000

[5] Gómez, Zeus. *Disseny i implementació en FPGA d'un computador SISA i del sistema operatiu ZeOS per a aquesta arquitectura.* (Juny de 2008). http://upcommons.upc.edu/pfc/handle/2099.1/5777

[6] Navarro, Juan J., Juan, Ton i Espasa, Roger. *Simple Instruction Set Architecture-3*

[7] Patterson, David i Hennessy John. *Computer organization and design. The hardware/software interface*. Capítol 4.

[8] Mulder, Karel. *Desarrollo sostenible para ingenieros.* Capítol 4.

[9] Mulder, Karel. *Desarrollo sostenible para ingenieros.* Capítol 10.

[10] *std_logic .* http://www.cs.sfu.ca/~ggbaker/reference/std_logic/copyright.html

[11] *std_numeric.* http://www.eda.org/rassp/vhdl/models/standards/numeric_std.vhd

[12] *DE1 User's Manual.*

[13] *Cyclone II Device Handbook.*

[14] *WEEE Directive*.
http://europa.eu/legislation_summaries/environment/waste_management/l21210_en.htm

Annex A. Assoliment de les competències tècniques

Aquest projecte ha estat elaborat en el context del Treball Final de Grau del pla d'estudis del Grau en Enginyeria Informàtica de la Facultat d'Informàtica de Barcelona, implantat al 2010. A l'inscriure el projecte, s'insta a l'estudiant a seleccionar un seguit de competències tècniques d'especialitat que vol aprofundir durant el treball. L'annex present és la justificació d'aquest desenvolupament.
Les competències triades per aquest projecte van ser:

CEC1.2: Dissenyar/configurar un circuit integrat utilitzant les eines de software adients. [Grau de profunditat: bastant]

Gran part del projecte es basa en el disseny del processador i la programació a la FPGA. Descobrir eines que ho aconsegueixin de manera fàcil i ràpida és una prioritat inicial del treball, ja que es treballarà amb elles durant tot el procés. S'ha fet servir el programa Quartus II i s'ha vist ideal per a aquest projecte, ja que incloïa totes les eines necessàries per al desenvolupament i feia la transició entre tasques (escriptura i simulació, o simulació i depuració) molt fluïda.

CEC2.1: Analitzar, avaluar, seleccionar i configurar plataformes hardware per al desenvolupament i l'execució d'aplicacions i serveis informàtics. [Grau de profunditat: una mica] i **CEC3.1:** Analitzar, avaluar i seleccionar les plataformes hardware i software més adients per al suport d'aplicacions encastades i de temps real. [Grau de profunditat: una mica]

L'anàlisi del dispositiu on es programarà el processador també és una tasca important al començament del projecte, però se li dedica menys importància perquè la gran majoria del temps s'està dissenyant i simulant, enlloc d'executar al dispositiu final. El motiu de l'elecció de la placa de desenvolupament DE1 d'Altera està detallada a l'apartat 1.5.

CEC3.2: Desenvolupar processadors específics i sistemes encastats; desenvolupar i optimitzar el software d'aquests sistemes. [Grau de profunditat: en profunditat]

Aquesta competència és la que millor descriu els objectius del projecte i, per això, és la que es treballa més. Tot el que es fa al projecte és per arribar a desenvolupar un processador específic, un SISA-3.

Annex B: Glossari d'abreviatures

AC2: Arquitectura de Computadors 2.

ACV: Anàlisi del Cicle de Vida.

ALU: Arithmetic Logic Unit.

ASIC: Application-Specific Integrated Circuit.

CMPS: Comparació en un Salt.

CPI: Cicles Per Instrucció.

DR: Decode and Read.

FPGA: Field-Programmable Gate Array.

GPL: GNU General Public License.

HDL: Hardware Description Language.

I/O: Input/Output.

LE: Logic Element.

LUT: Lookup Table.

MEM: Memory.

NOP: No Operation.

PAL: Programmable Array Logic.

PC: Program Counter.

PEC: Projecte d'Enginyeria de Computadors.

PLD: Programmable Logic Device.

RTL: Register-Transfer Level.

RSI: Rutina de Servei d'Interrupcions.

RAW: Read After Write.

SISA: Simple Instruction Set Architecture.

TC: Temps de Cicle.

TLB: Translation Lookaside Buffer.

VHDL: VHSIC Hardware Description Language.

WAR: Write After Read.

WAW: Write After Write.

WB: Write-Back.

WEEE: Waste Electrical and Electronic Equipment.