Capítol 3

Disseny

Amb les modificacions que acabem de veure al Capítol anterior hem modificat significativament el pipeline que teníem inicialment. Hem aplicat totes les tècniques descrites, i això dóna com a resultat un processador significativament diferent del que teníem. Hem afegit algunes etapes que abans no hi eren i hem distribuït feina que abans estava concentrada en una única etapa.

A continuació enumerarem les etapes finals de les que consta el nostre pipeline, explicant-ne la seva funcionalitat de manera molt general. El comportament exacte que segueix cada etapa i el seu funcionament intern es pot trobar en el Capítol 4 on es tracta de manera més precisa la implementació de cada etapa i els seus components.

Les etapes, ordenades tal i com les creuen les instruccions, són:

- FETCH: És on es carreguen les noves instruccions que entren al pipeline. També s'encarrega de duur la gestió de la predicció de salts.

- DECO: És on es decodifiquen les instruccions per saber quin tipus d'instrucció és i quins són els registres (físics) on té els operands.

- QUEUE: És on s'emmagatzemen les instruccions fins que ja tenen tots els seus operands disponibles.

- PICK: És on escollim quina instrucció de les que hi ha a QUEUE pot

abandonar la Cua i seguir executant-se pel pipeline.

- BR: És on hi ha el Banc de Registres d'on obtenim les dades necessàries per executar els càlculs que farem a continuació.

- ALU: És on es fan realment tots els càlculs necessaris per obtindre el resultat de les operacions aritmètiques i lògiques. També és on es fan tots els càlculs relatius als salts. En el cas de les instruccions d'accés a memòria, és on es calcula la direcció de l'accés.
  Aquesta etapa té tres sub-pipelines (de 1, 4 i 7 etapes respectivament) per donar suport a les operacions simples, les multiplicacions i les divisions.

- CACHE: És on es fan els accessos a Memòria. També és on hi ha la Cache de Dades i on es gestionen els Store.

- WB: És on es fan les escriptures dels resultats d'etapes anteriors al Banc de Registres. També és on hi ha la GL+HF i on es fa tota la seva gestió.

A la Figura 3.1 podem veure com queda finalment el nostre pipeline. Veiem que hem ampliat la seva llargada degut a la inclusió de noves etapes. Això fa que les instruccions segueixin un camí diferent al que seguien en el SISA-S original. A continuació farem una descripció detallada del recorregut que segueix cada tipus d'instrucció al seu interior, des del moment en que la instrucció entra a FETCH fins que fa Commit a WB. L'últim cicle, Commit, no es mostra en els esquemes gràfics, doncs la instrucció simplement s'invalida.

Agrupem les instruccions que tenen una funcionalitat semblant. Aquests grups coincideixen amb els que es poden veure a la taula d'instruccions (Figura 2.2). Aquelles instruccions que segueixen una pauta diferent de les del seu grup són tractades apart. Per a cada tipus d'instrucció mostrarem quin és el recorregut que fan pel pipeline i la seva latència (període de temps entre el moment d'entrar al processador i el moment de sortir-ne). Aquest temps el donarem en cicles, ja que per calcular-ho en segons necessitaríem el temps de cicle, i és una dada que desconeixem.

Totes les dades de latències es donen suposant que la instrucció no es

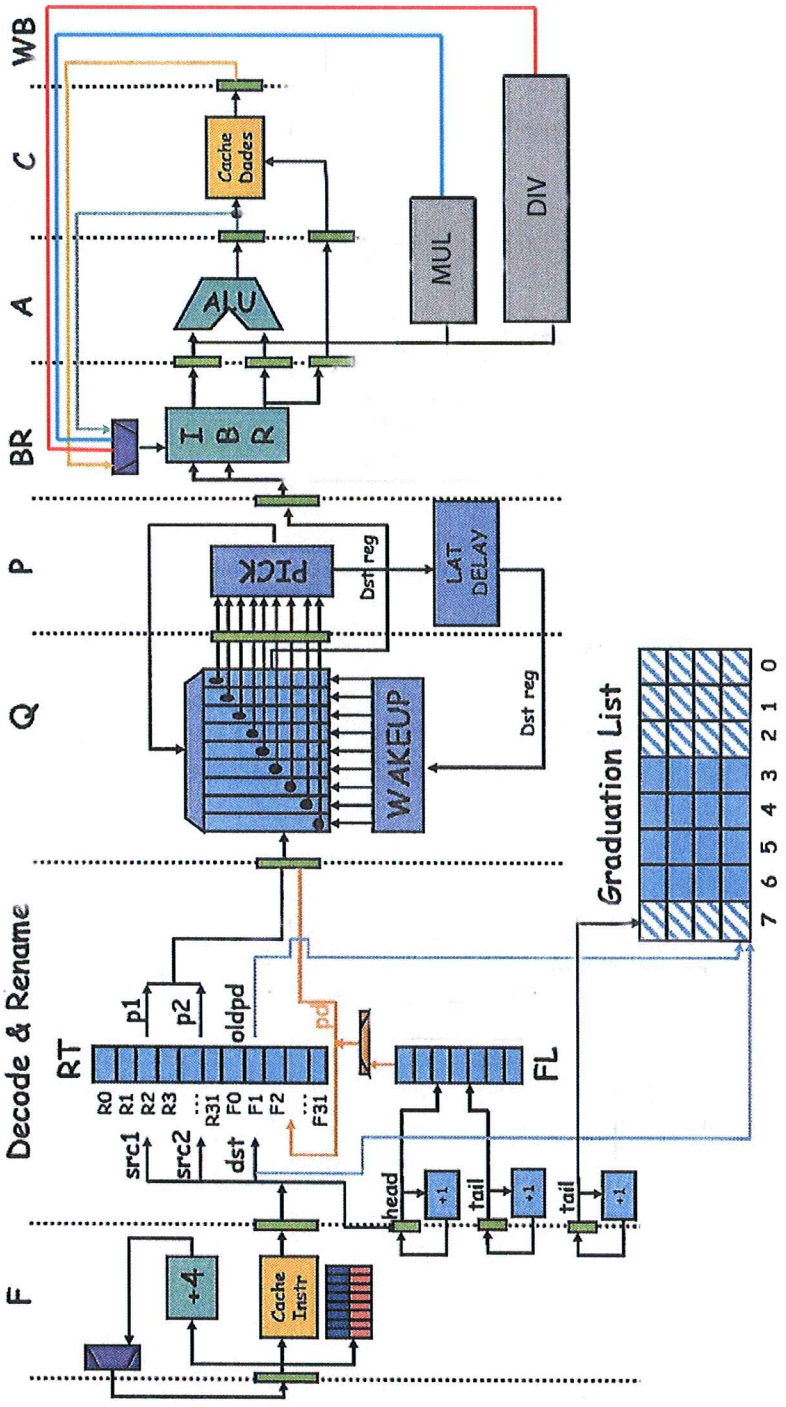

Figura 3.1: Pipeline sencer

bloqueja pel pipeline, sinó que s'executa en solitari sense interferències de cap altra instrucció. Per tant, estem donant la *latència ideal.*

3.1. Inst. Aritmètiques i lògiques

Dins d'aquesta categoria hi trobem les següents instruccions

- ADD, SUB, SHA, SHL
- AND, OR, XOR, NOT

Totes les instruccions d'aquest grup realitzen operacions aritmètiques entre dos operands agafats del Banc de Registres d'Usuari i dipositen el resultat en un registre del mateix Banc. Les instruccions entren per FETCH, es decodifiquen els seus operands a DECO i entren a la Cua de QUEUE. En el moment en que PICK les sel·lecciona, passen a BR per agafar els seus operands i, un cop ALU ha calculat el seu resultat, passen directament cap a WB que escriu el resultat al Banc de Registres. En el mateix cicle informem al GL+HF de que ja hem acabat, i només ens queda esperar a que es faci el Commit.

FETCH → DECO → QUEUE → PICK → BR → ALU → WB → Commit

Ho podem veure gràficament a la Figura 3.2. En total, aquest tipus d'instruccions triguen 8 cicles en executar-se, des del moment en que entren a FETCH fins que fan Commit.

3.2. Inst. de Comparació

Les instruccions que entren dins d'aquesta categoria són:

- CMPLT
- CMPLE

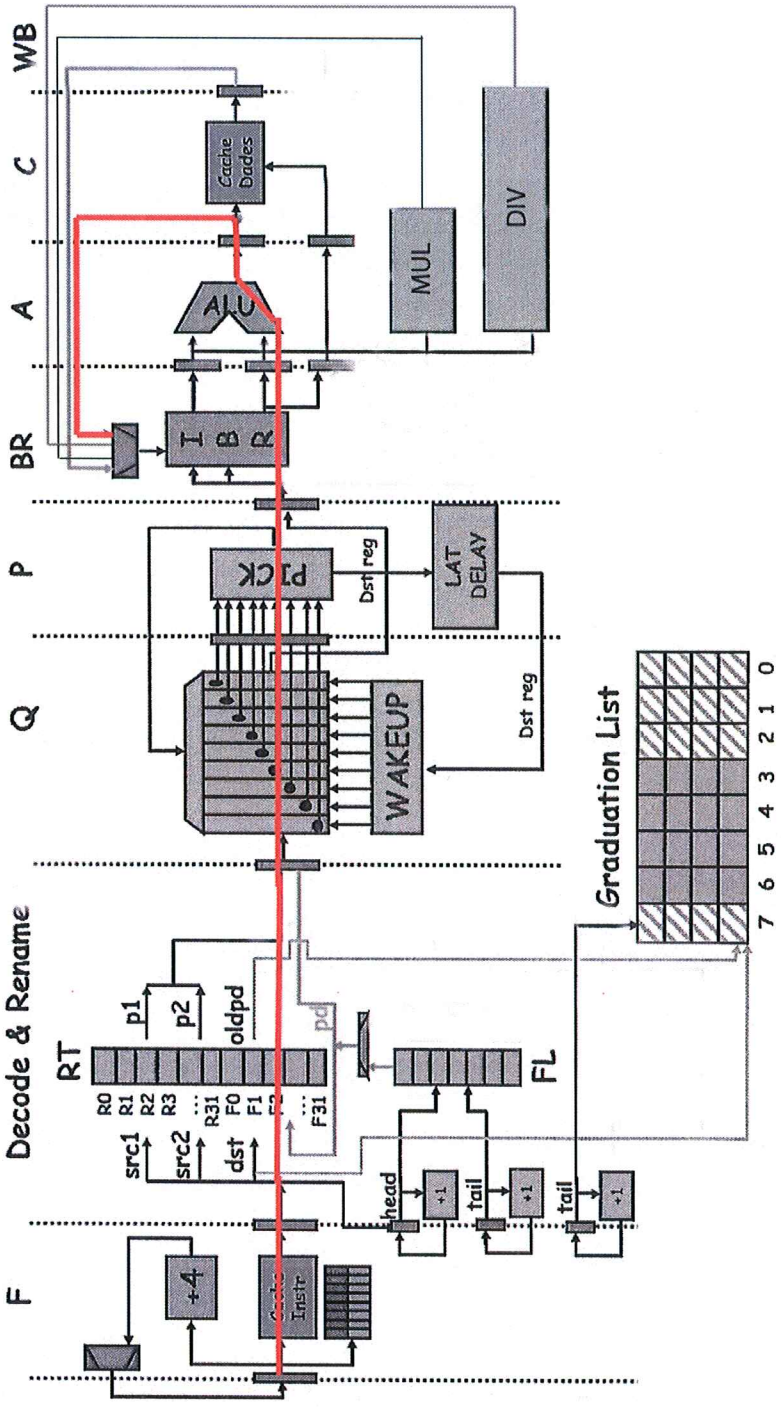

Figura 3.2: Pipeline de les instruccions aritmètiques

- CMPEQ
- CMPLTU
- CMPLEU

Les instruccions d'aquest grup fan una comparació entre dos registres, i segueixen exactament el mateix recorregut que les del grup anterior. Un cop han fet la part comuna de totes les etapes (FETCH, DECO, QUEUE, PICK, BR) entren a ALU i fan els càlculs de comparació corresponents. Després passen a WB per escriure el resultat al Banc de Registres d'Usuari i marcar la instrucció com a *Acabada*. Finalment, fem Commit en el moment en que sigui la instrucció més antiga del pipeline.

FETCH → DECO → QUEUE → PICK → BR → ALU → WB → Commit

La Figura 3.3 mostra aquest comportament. Es tracta del mateix que pel cas anterior, ja que una comparació no deixa de ser una resta on es comproven els bits de control, per això fan el mateix recorregut pel pipeline. En total, aquest tipus d'instruccions triguen 8 cicles en executar-se, des del moment en que entren a FETCH fins que fan Commit.

3.3. Instruccions amb un operador immediat

Les instruccions que podem englobar en aquest grup són:

- ADDI
- MOVI
- MOVIH

Una instrucció amb immediat segueix exactament la mateixa filosofia que les instruccions aritmètiques (veure Capítol 3.1), amb la única diferència de que algun dels seus operands s'extreu del Banc de Registres d'Usuari. L'altre operand és un valor concret que s'extreu directament del codi de la instrucció (IR).

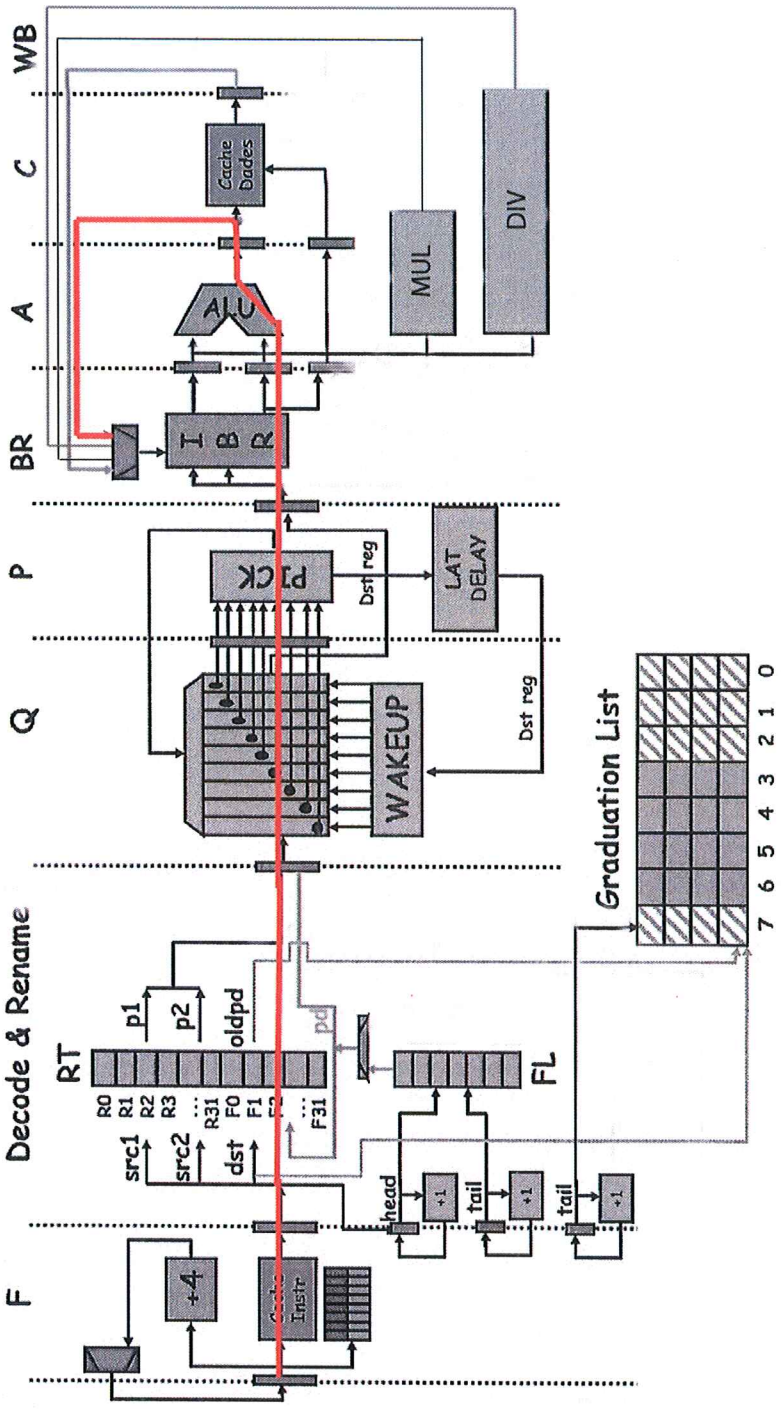

Figura 3.3: Pipeline de les instruccions de comparació

El conjunt d'etapes que recorre és el mateix. En total triga 8 cicles en executar-se completament.

3.4. Ruptura de seqüència simple

El comportament normal del processador estableix que un cop executada una instrucció intentem executar la que hi ha a continuació. Tot i així hi ha maneres d'indicar al processador que cal variar aquest comportament i que volem executar altra instrucció diferent. Això s'anomena "ruptura de seqüència", i les instruccions que permeten fer-ho són:

- JZ, JNZ, JMP
- BZ, BNZ

Les hem anomentat "simples" perquè totes elles segueixen una pauta (sempre la mateixa) d'interacció amb els mòduls que gestionen els salts, sense interactuar amb res més (JAL i CALL no ho compleixen), i són totes instruccions d'Usuari (RETI serà tractat junt amb les instruccions de Sistema).

Aquestes instruccions fan el recorregut següent: un cop ja han entrat al pipeline i han arribat fins a la Cua (FETCH, DECO, QUEUE, PICK), accedeixen als registres de BR (Rx) per agafar els seus operands. Amb aquests valors arriben fins a ALU, on calculen les condicions del salt per saber si cal informar d'un error o no. Sigui com sigui, un cop han arribat a ALU ja ha acabat la seva execució, de manera que també informen a GL+HF de que han acabat. Finalment, faran Commit quan siguin la instrucció més vella.

FETCH → DECO → QUEUE → PICK → BR → ALU → Commit

Podem veure aquest comportament en la Figura 3.4. En total, una instrucció de salt simple triga 7 cicles en executar-se.

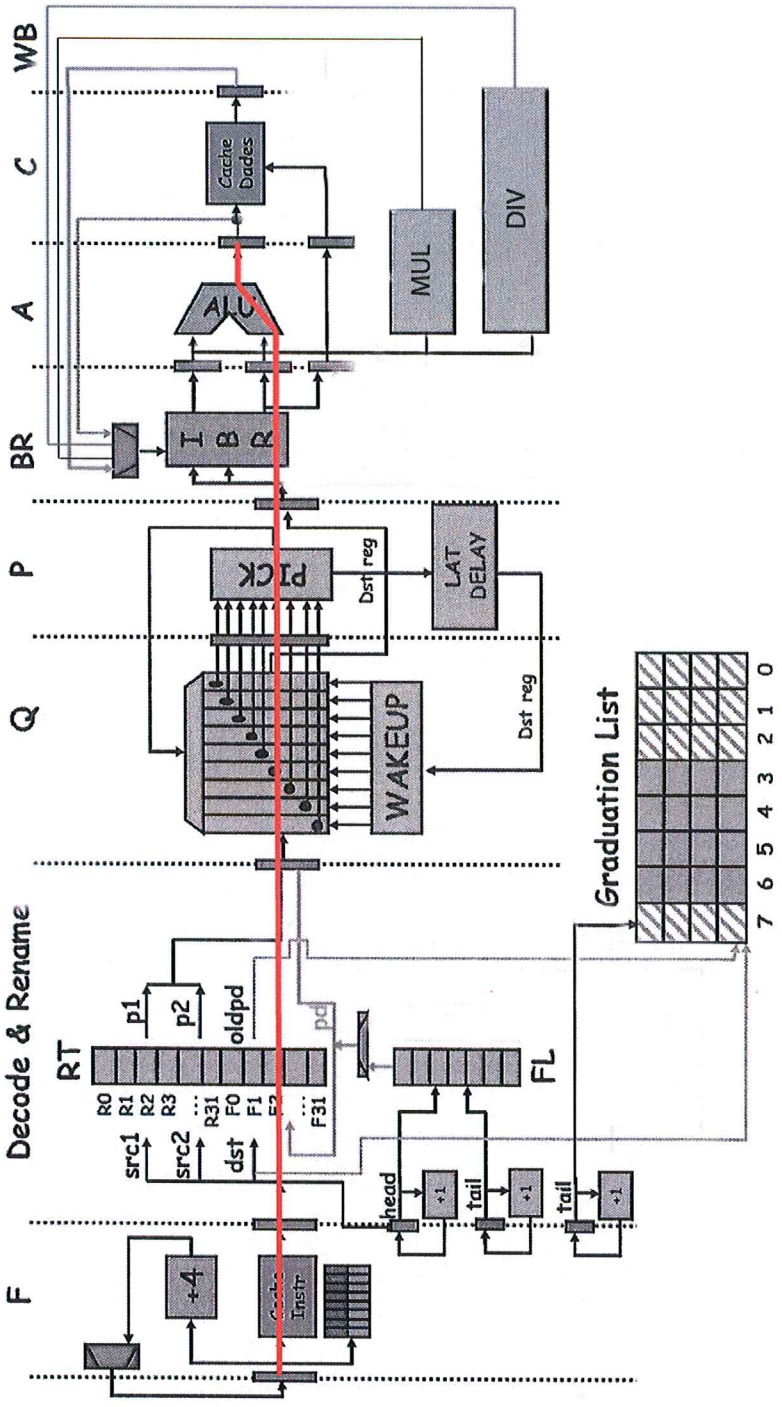

Figura 3.4: Pipeline de les instruccions de trencament de seqüència simple

3.5. Instrucció de salt JAL

La instrucció JAL és una instrucció de salt que no segueix el mateix esquema que els salts que acabem de descriure. Un JAL és una instrucció que, apart de carregar el valor d'un operand com a PC, també fa una escriptura al Banc de Registres d'Usuari per deixar-hi el valor del PC següent de la instrucció. Aquest detall fa que en realitat es tracti d'una barreja entre una operació aritmètica i un salt simple.

FETCH → DECO → QUEUE → PICK → BR → ALU → WB → Commit.

En total, un JAL triga 8 cicles en executar-se completament. La Figura 3.5 mostra aquest comportament.

3.6. Instrucció d'entrada al Sistema Operatiu CALL

La instrucció CALL és, juntament amb JAL, l'altra instrucció de salt que no segueix la pauta dels salts simples. En aquest cas, la característica que el fa diferent és que té interacció amb el Banc de Registres de Sistema, ja que en modifica el valor de S0, S1 i S7.

Tal i com ja hem explicat, el Renombrament i el comportament "Fora d'Ordre" només els apliquem al Banc de Registres d'Usuari (Rx), de manera que si una instrucció modifica Sx cal buidar el pipeline per tornar a començar. En aquest cas, el CALL segueix el comportament dels salts simples fins que arriba a ALU. En aquest punt informa al GL+HF de que ja ha acabat la seva execució. En el moment en que sigui la instrucció més vella del pipeline en fem el Commit, i això vol dir que tots els càlculs i l'escriptura de resultats a Sx es fan en aquest moment. Mentre es duu a terme aquesta escriptura s'envien senyals per buidar tot el pipeline i començar de nou per la primera instrucció del Sistema Operatiu (valor que hem agafat de S5).

FETCH → DECO → QUEUE → PICK → BR → ALU → Commit

Podem veure aquest comportament en la Figura 3.6. En total, un CALL triga 7 cicles en arribar a Commit.

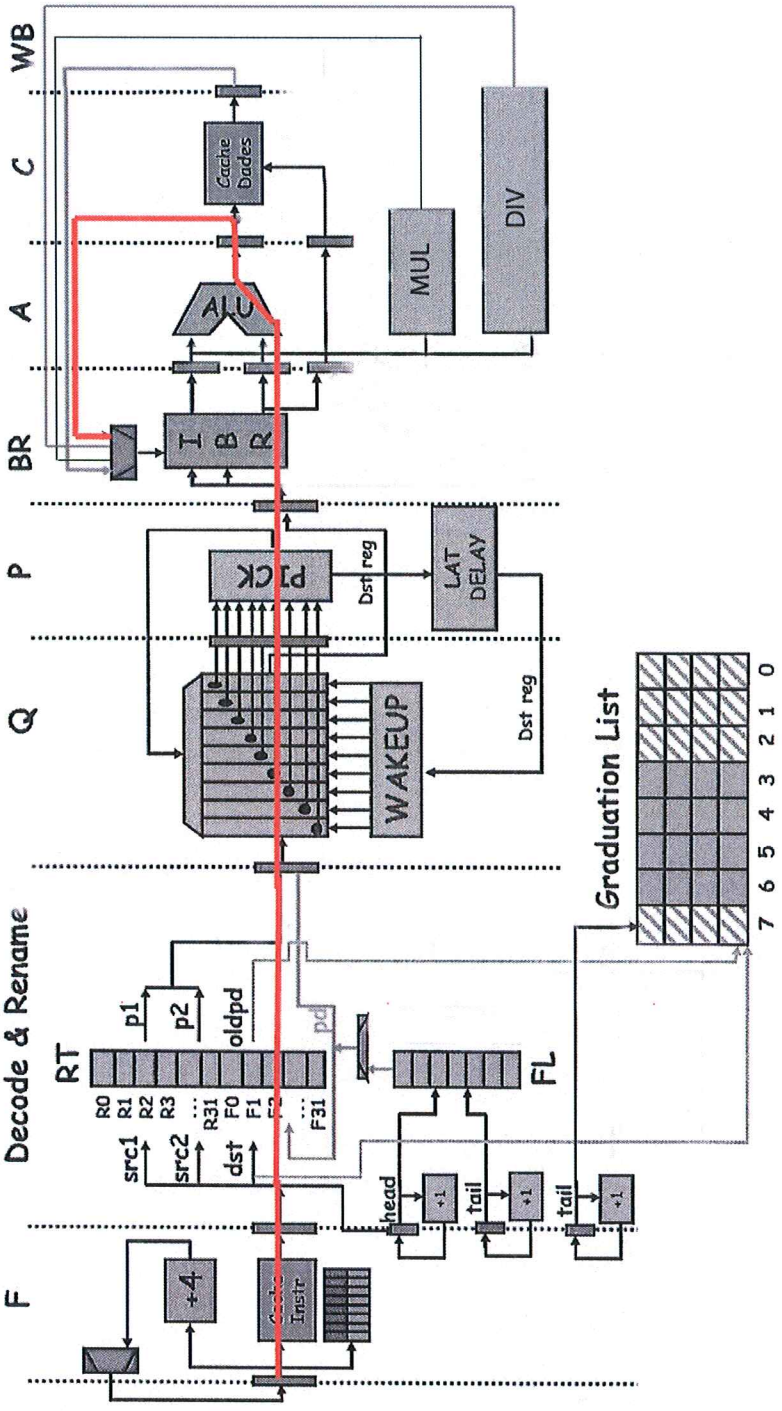

Figura 3.5: Pipeline de JAL

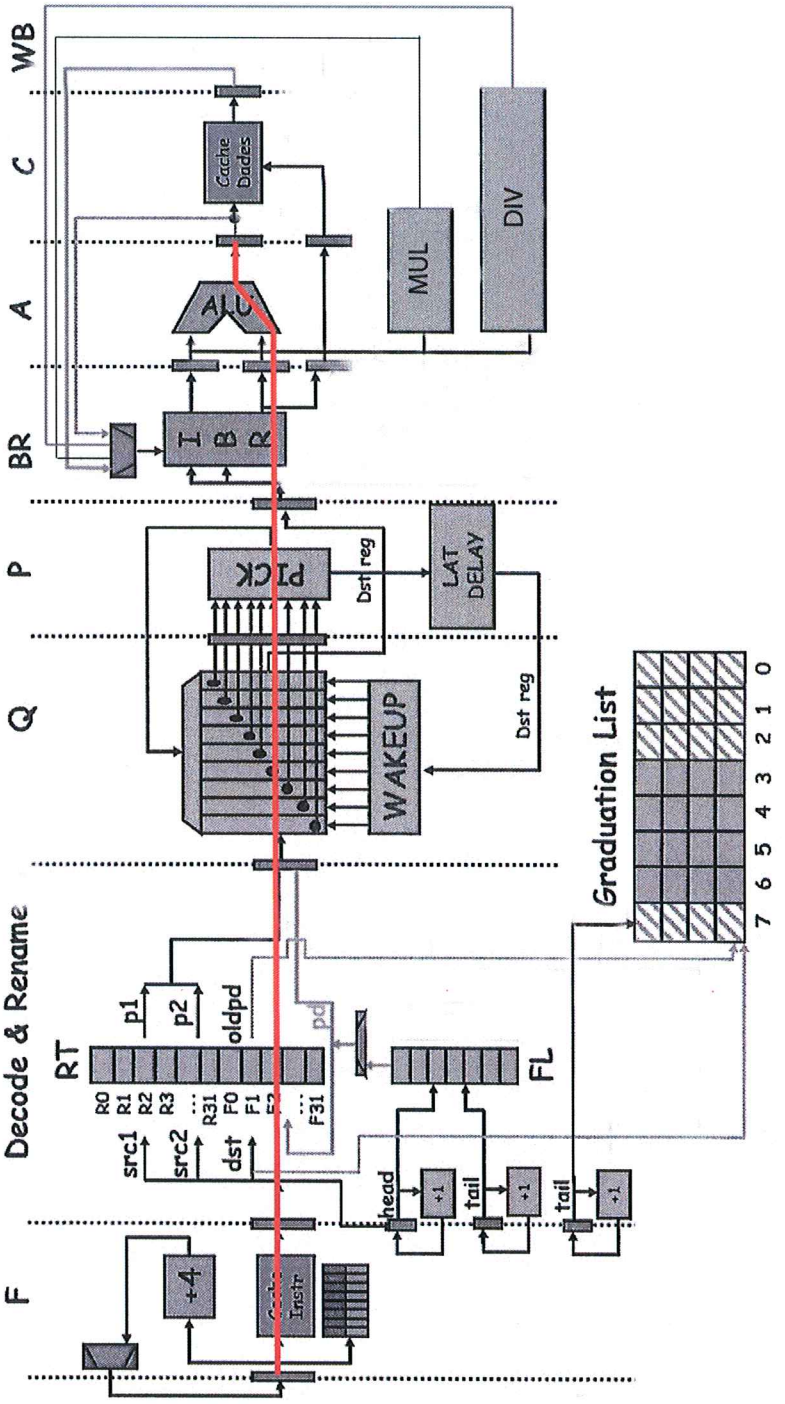

Figura 3.6: Pipeline de CALL

3.7. Loads

La instrucció Load s'encarrega de portar una dada des de la memòria cap al processador. Tenim dos tipus diferents de Load:

- LD
- LDB

El primer tracta amb posicions de memòria de tamany Word (2 Bytes, 16 bits), mentre que el segon tracta directament amb dades de tamany Byte. Quan un Load (del tipus que sigui) ha recorregut les etapes comunes (FETCH, DECO, QUEUE, PICK), obté el seu operand del Banc de Registres d'Usuari i passa a ALU, on es fan tots els càlculs relatius a la direcció d'accés. Llavors entra dins de CACHE, on obtenim les dades, i passa a WB per escriure-la al Banc de Registres d'Usuari.

FETCH → DECO → QUEUE → PICK → BR → ALU → CACHE → WB → Commit

Ho podem veure en la Figura 3.7. En total, si un Load té la dada disponible a CACHE triga 9 cicles en fer el Commit. Si la dada no es troba a CACHE, cal demanar-la al Bus, que trigarà un nombre indeterminat de cicles en respondre; llavors ja no podem estimar una xifra de cicles consumits.

3.8. Stores

La instrucció Store fa el pas invers al Load: treu una dada de les que té el processador i l'envia cap a memòria. Igual que ha passat amb els Load, tenim dos instruccions diferents de Store en funció del tamany de les dades que tracta:

- ST
- STB

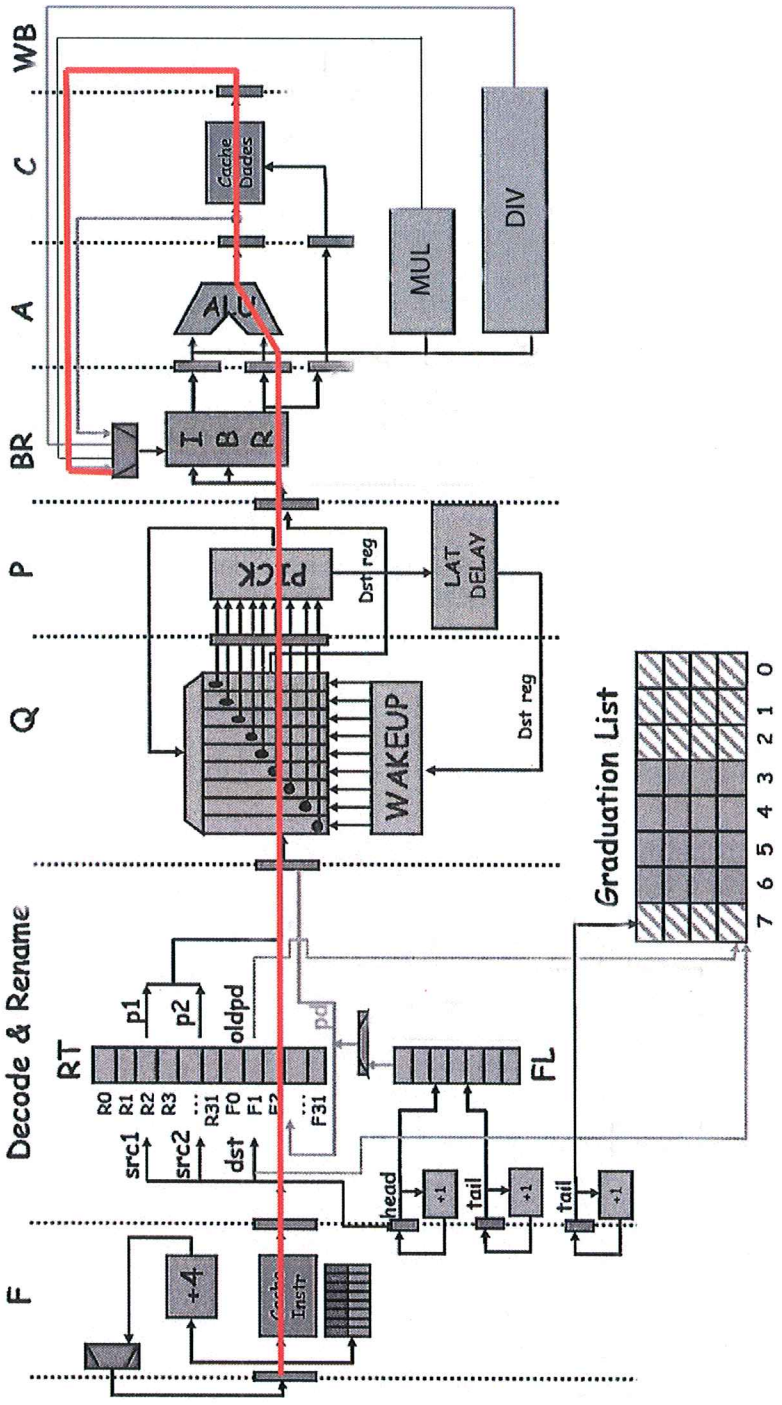

Figura 3.7: Pipeline de les instruccions LOAD

El primer escriu 1 Word (2 Bytes) a memòria, mentre que el segon escriu 1 Byte. En aquest cas volem escriure una dada, però fora del processador. Un cop hem arribat fins a BR accedim al valor dels operands i passem a ALU on (igual que passava amb els Loads) preparem la direcció d'accés i la dada que volem escriure. Llavors passem a CACHE, on la instrucció es queda emmagatzemada dins del **StoreBuffer**, i indiquem al GL+HF que la instrucció ja ha acabat. En el moment en que li toqui fer el Commit marquem aquest Store com a *Preparat* i ja ens en desentenem. Quan pugui anirà a escriure's, però ja no té més efectes de cara al pipeline ni les instruccions següents. Aquesta diferència fa que ens estalviem una etapa (WB), ja que tots els cicles que es poden consumir des del moment de fer Commit fins que l'escriptura es fa efectiva a memòria no compten a nivell de retard.

FETCH → DECO → QUEUE → PICK → BR → ALU → CACHE → Commit

Veiem a la Figura 3.8 el diagrama corresponent a aquest comportament. En total, una instrucció de Store triga 8 cicles en fer el seu Commit.

3.9. Enviar una dada cap a l'espai d'Entrada/Sortida (IN)

La instrucció IN és un cas d'instrucció on no podem predir el seu cost en cicles. A diferència dels Load, on sabem el seu cost en cas de *Hit* (la dada està a la Cache) però no en cas de *Miss* (la dada no està a la Cache), un IN sempre necessita accedir a l'exterior del pipeline per portar una dada. Aquest fet fa que no sigui possible estimar una quantitat de cicles. Quan un IN ha arribat fins a ALU entra dins l'estructura (FIFO) de IN/OUT, i es queda allà emmagatzemada fins que obté la dada. Llavors passa directament cap a WB per escriure-la al Banc de Registres d'Usuari i informar al GL+HF de que ja ha acabat. Quan sigui la més vella farà Commit.

FETCH → DECO → QUEUE → PICK → BR → ALU → ... → WB → Commit

Aquesta instrucció, junt amb les altres dues que també interaccionen

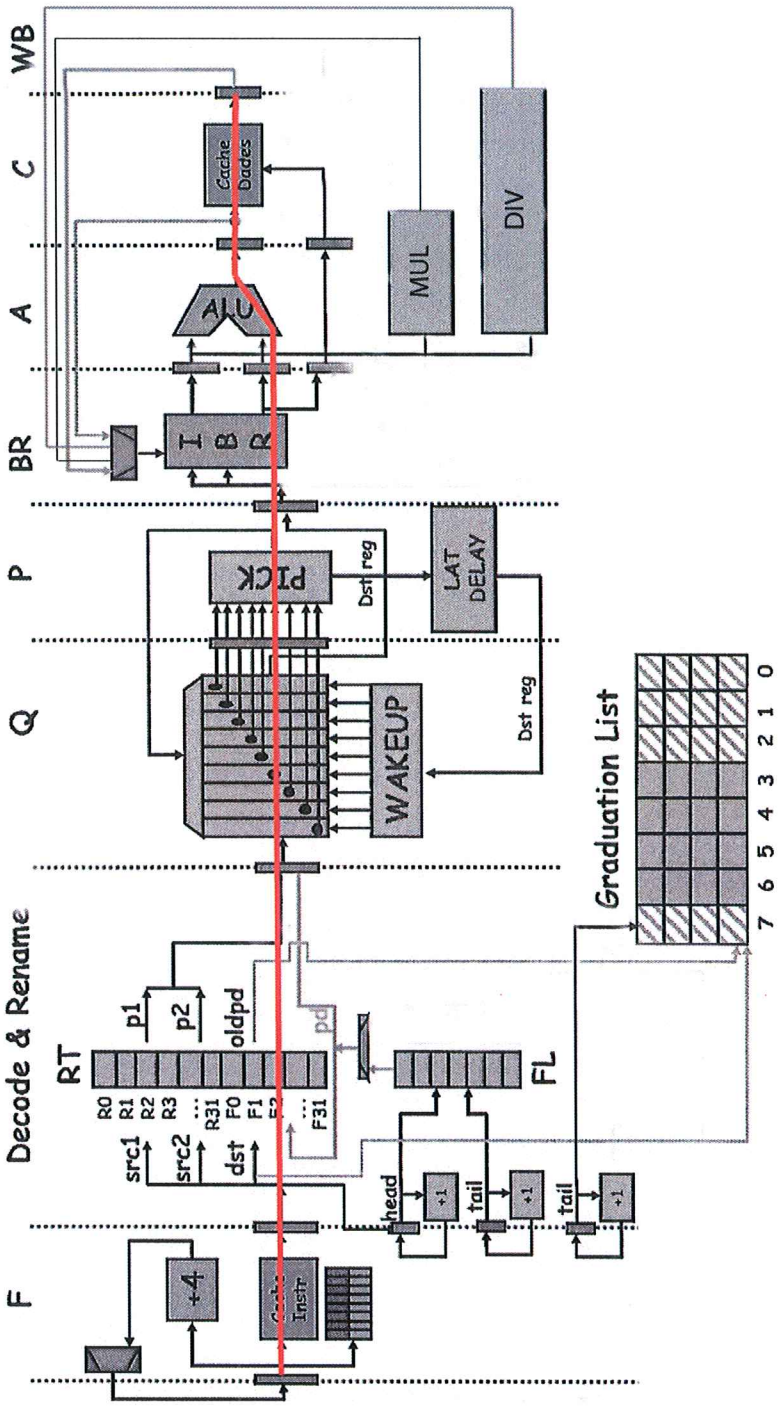

Figura 3.8: Pipeline de les instruccions STORE

amb l'Espai d'Entrada/Sortida (OUT i GETIID) són instruccions que entren a executar-se en ordre, arribant sempre a ALU en el mateix ordre en que han entrat al processador.

El diagrama és el mateix que tenim per a les instruccions aritmètiques simples, amb la diferència que ens quedem esperant a ALU una quantitat indeterminada de cicles. No podem donar una aproximació del cost.

3.10. Rebre una dada de l'espai d'Entrada/Sortida (OUT)

A diferència del que succeeix amb IN, en aquest cas sí que podem calcular quants cicles trigarà en executar-se un OUT. En realitat segueix el mateix funcionament que els Stores, amb la diferència de que no s'acumulen a CACHE, sinó a ALU. Quan un OUT arriba a ALU, aquest s'encua dins l'estructura de IN/OUT, però informa al GL+HF de que ja ha acabat. Quan sigui la instrucció més vella, llavors marquem aquest OUT com a *Preparat*, i en fem el Commit.

FETCH → DECO → QUEUE → PICK → BR → ALU → Commit

El diagrama per veure el seu comportament és el mateix que en el cas de les instruccions de salt (Figura 3.9). En total, un OUT triga 7 cicles en fer Commit. Tots els cicles que tardi després en fer efectiva l'escriptura ja no entren dins del cost en cicles ja que no afecta a la resta d'instruccions.

3.11. Multiplicacions

Les intruccions de multiplicació són:

- MUL
- MULH
- MULHU

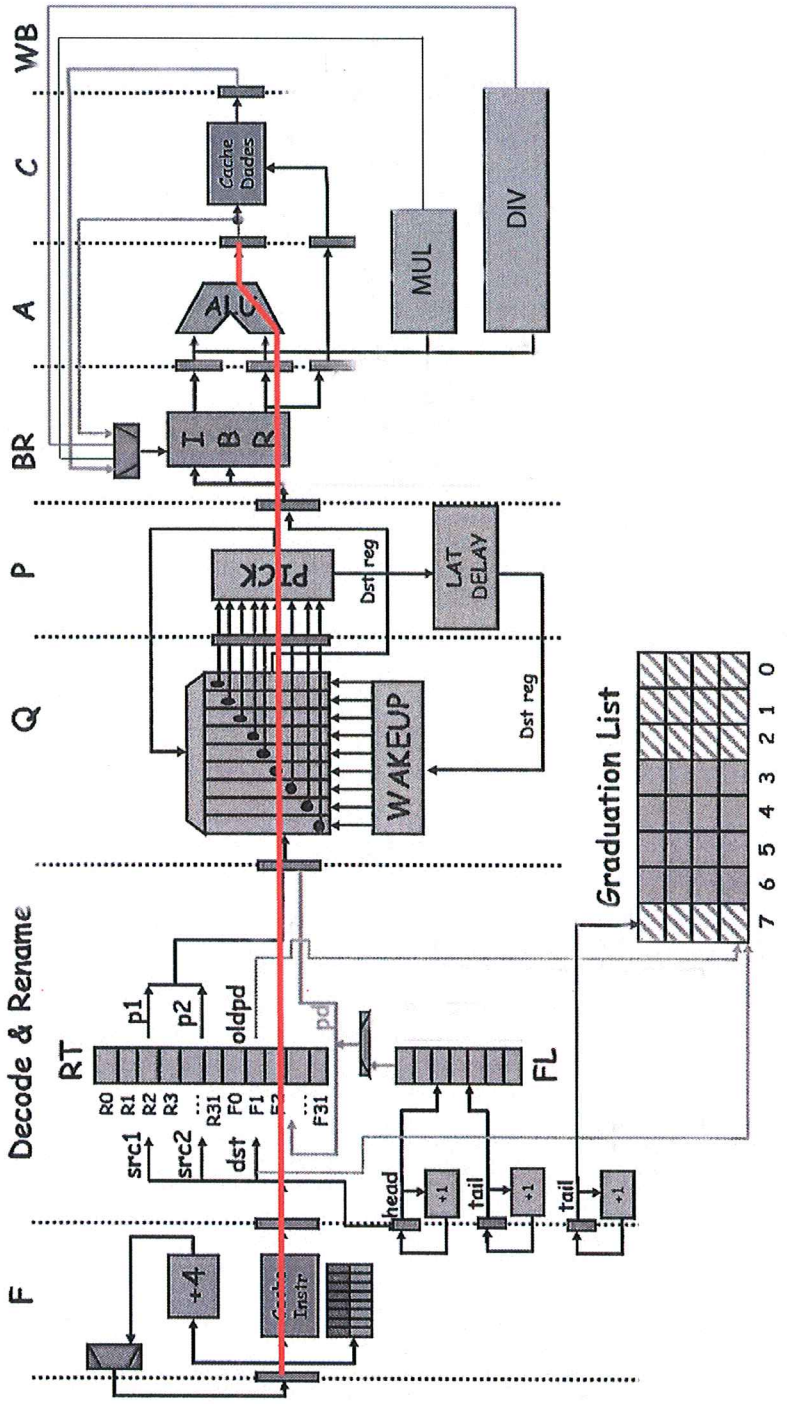

Figura 3.9: Pipeline de OUT

Les tres instruccions, tot i donar resultats potencialment diferents, tenen exactament el mateix comportament dins el pipeline. Realment es tracta d'un cas d'instrucció aritmètica però que, enlloc d'estar 1 sol cicle a ALU n'hi està 4; triga 4 cicles dins el mòdul de ALU destinat a les multiplicacions, i llavors passa cap WB per escriure el seu resultat al Banc de Registres d'Usuari. Excepte els 4 cicles dins de ALU, la resta del recorregut és exactament el mateix que el de les operacions aritmètiques simples.

FETCH → DECO → QUEUE → PICK → BR → ALU1 → ALU2 → ALU3 → ALU4 → WB → Commit

Les mulplicacions, tal i com ja hem comentat, fan servir un pipeline de 4 etapes segmentades, fet que fa possible tenir més d'una multiplicació alhora dins de ALU.

El diagrama que reflexa el comportament de les multiplicacions es troba a la Figura 3.10. En total, una instrucció de multiplicació triga 11 cicles en arribar fins al seu Commit.

3.12. Divisions

Les instruccions de divisió són:

- DIV
- DIVU

Amb les divisions succeeix un cas semblant al de les multiplicacions, i les dues instruccions de divisió tenen el mateix comportament. Un cop arriba una divisió a ALU, aquesta triga 7 cicles en tenir calculat el resultat per poder seguir fins a WB i escriure el resultat al Banc de Registres d'Usuari.

FETCH → DECO → QUEUE → PICK → BR → ALU1 → ALU2 → ALU3 → ... → ALU7 → WB → Commit

A diferència de les multiplicacions, el sub-pipeline de ALU de 7 etapes no està segmentat, de manera que no podem tenir dues o més divisions alhora dins de ALU.

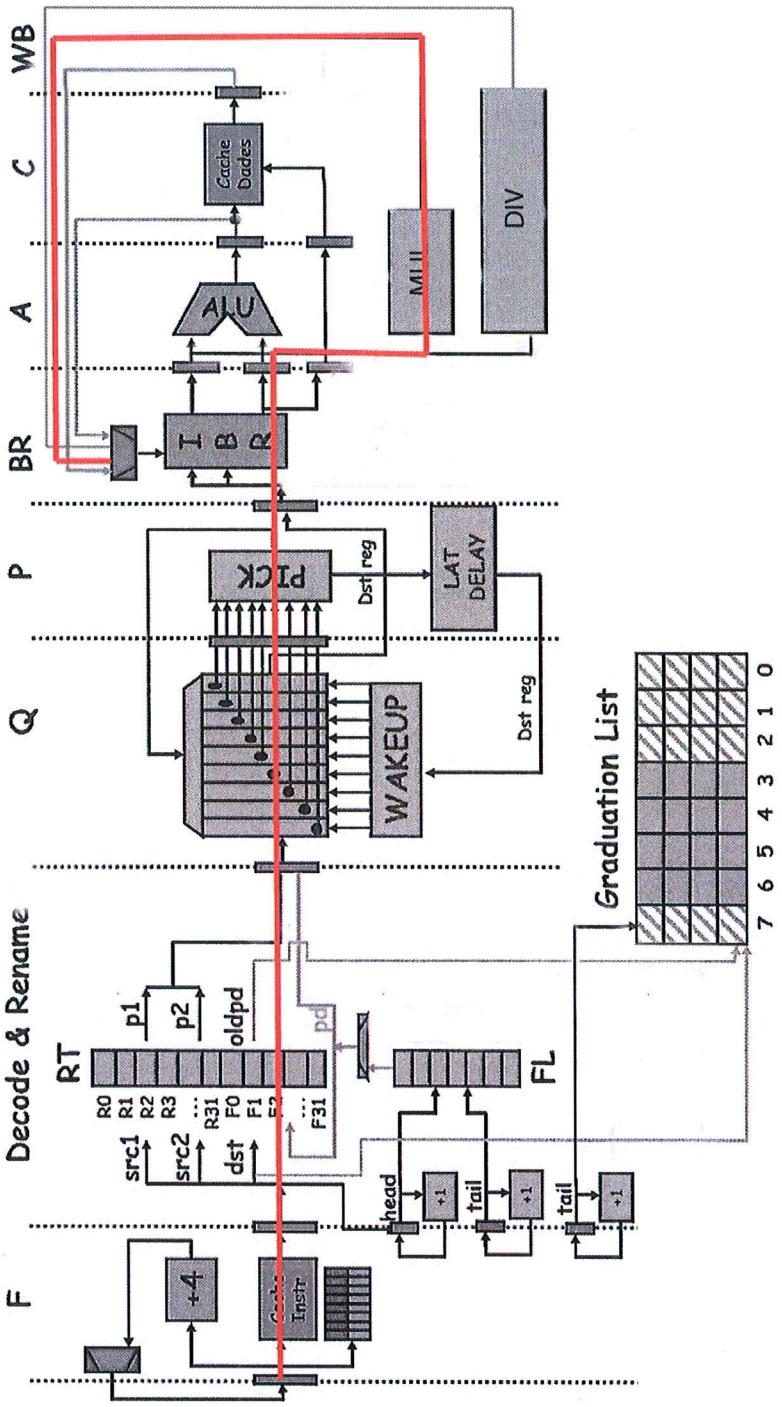

Figura 3.10: Pipeline de les instruccions de multiplicació

El diagrama que mostra el comportament de les divisions és una extensió del de les multiplicacions (Figura 3.11). En total, una instrucció de divisió triga 14 cicles en acabar la seva execució i sortir del processador (Commit).

3.13. Inst. de Sistema

Les instruccions que tractem en aquest apartat són:

- EI
- DI
- RETI
- WRS
- WTLBI
- WTLBD
- FLUSH
- HALT

Excepte un parell de casos que comentarem en els punts següents, totes les instruccions de sistema tenen el mateix comportament. Totes elles necessiten escriure un resultat al Banc de Registres de Sistema, i això només ho permetem fer en el moment del Commit. Per això, totes elles arriben fins a l'etapa de ALU, on preparen els operands que necessitaran per escriure i, directament, informen al GL+HF de que ja han acabat. Quan els arriba el torn de fer Commit, els valors que volen escriure (i que tenen emmagtzemats al GL+HF) surten i es van a escriure on correspongui (Banc de Registres de Sistema, TLB o Cache) segons el cas.

Totes aquestes instruccions necessiten que en el moment de fer Commit es buidi el pipeline i es torni a començar per la instrucció següent a elles (excepte RETI, que carrega un valor concret com nou PC).

FETCH → DECO → QUEUE → PICK → BR → ALU → Commit

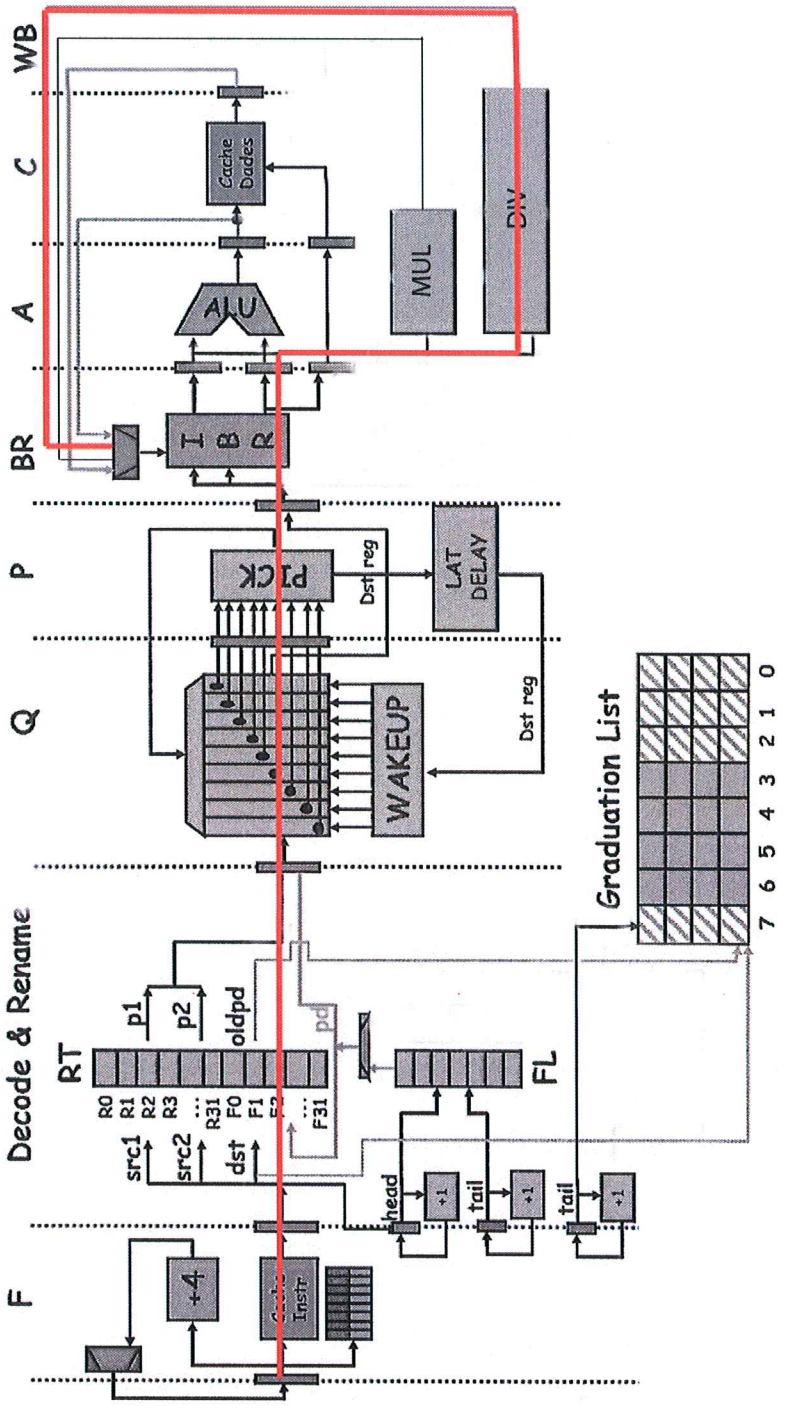

Figura 3.11: Pipeline de les instruccions de divisió

El diagrama que mostra aquest comportament és el mateix que els de les instruccions de salt simples (Figura 3.12). En total, una instrucció de Sistema (de les que hem esmentat) triga 7 cicles en executar-se completament.

3.14. Obtenir l'identificador d'interrupció entrant (GETIID)

GETIID no és més que un cas concret d'una instrucció IN (veure Capítol 3.9), amb la diferència que no volem una dada qualsevol, sinó que sempre demanem l'identificador de la interrupció que ha arribat. En aquest cas es tracta d'una instrucció de Sistema i caldrà fer algunes comprovacions durant la implementació interna però, a nivell de comportament, el seu recorregut pel pipeline és exactament el mateix que el que tenim en una instrucció IN, inclosa l'escriptura del resultat al Banc de Registres d'Usuari.

3.15. Llegir del Banc de Registres de Sistema (RDS)

La instrucció RDS és un cas d'instrucció de Sistema que accedeix al Banc de Registres de Sistema per llegir un operand, però que escriu el resultat al Banc de Registres d'Usuari. Aquest fet fa que, tot i que és una operació de Sistema, realment tingui el mateix comportament que una instrucció aritmètica. Un cop la instrucció ha sortit de QUEUE, aquesta accedeix al Banc de Registres de Sistema i n'extreu un operand. Quan arriba a ALU realment no tracta aquest valor, sinó que el deixa intacte per a ésser el resultat. Llavors passa a WB, des d'on escriu el valor al Banc de Registres d'Usuari i informa al GL+HF de que ja ha acabat la seva execució. Quan sigui la instrucció més vella del pipeline farà Commit.

Realment podem estar segurs de que el valor obtingut de Sx és correcte, ja que si alguna instrucció més vella modifica posteriorment aquest valor, es tractarà d'una instrucció de Sistema que buidarà el pipeline, inclòs el RDS actual.

FETCH → DECO → QUEUE → PICK → BR → ALU → WB → Commit

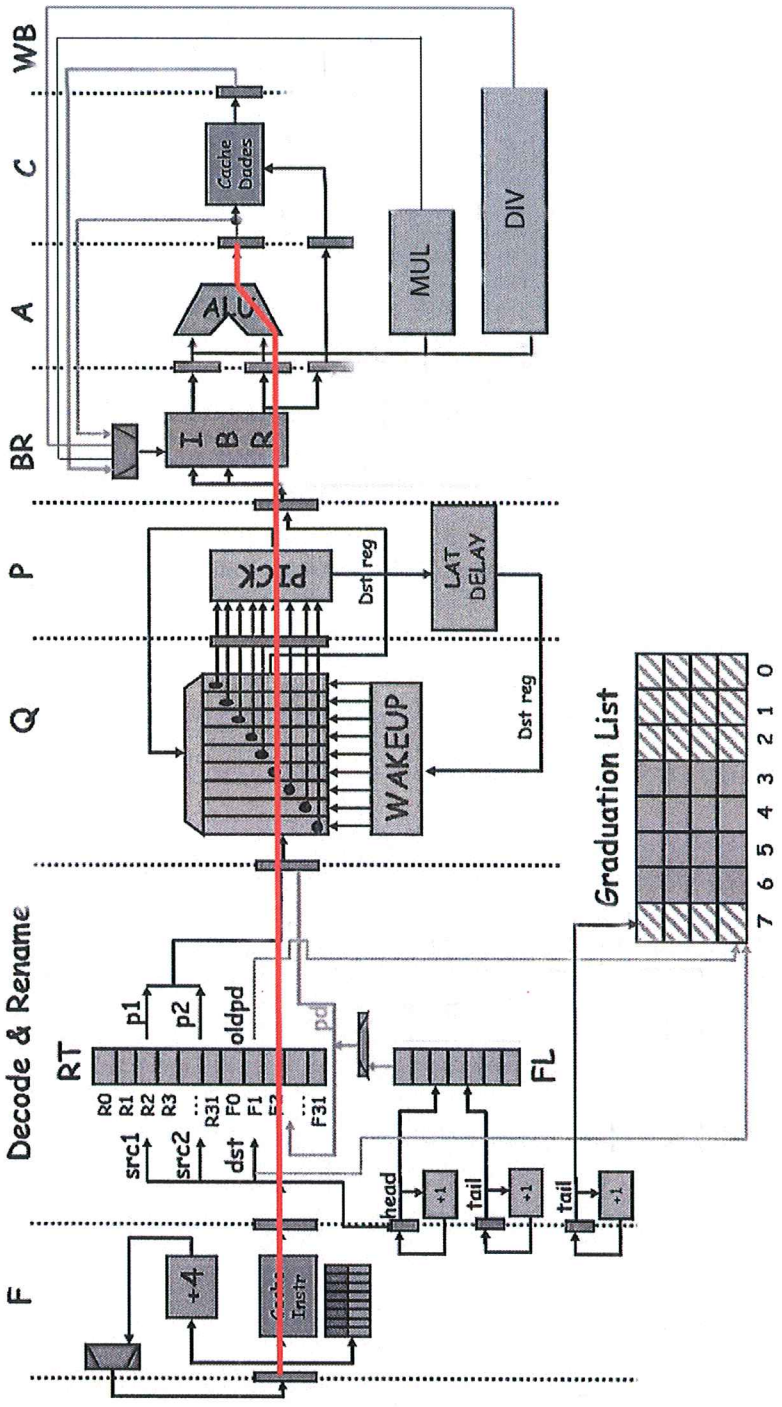

Figura 3.12: Pipeline de les instruccions de sistema

El diagrama del seu comportament és el mateix que el de les instruccions aritmètiques (Figura 3.13). En total, una instrucció RDS triga 8 cicles en executar-se completament i fer el Commit per sortir del pipeline.

3.16. Tamany de les dades

Dins l'etapa de disseny també hi podem incloure totes les decisions que fan referència al tamany de les estructures. Totes les millores que acabem de presentar impliquen afegir elements de hardware on el seu tamany és un valor que podem decidir, i aquesta decisició tindrà un impacte directe amb el rendiment.

Aquestes són les dades numèriques que identifiquen els tamanys de les estructures:

- TLB: 8 entrades (7 de generals i 1 de reservada per al Sistema Operatiu).
- 2 Caches independents de 1KB, una a FETCH i una a CACHE. Tamany de línia: 16 Bytes. 64 entrades.
- Memòria Principal: 64MB. Pàgines de 1KB.
- Branch Predictor (BP): 16 entrades.
- Banc de Registres Enters (Rx): 32 registres físics de 16 bits.
- Banc de Registres Especials (Sx): 8 registres físics de 16 bits.
- Taula de Renaming: 8 entrades, una per a cada registre lògic de Rx.
- Free-List: 32 posicions totals, 8 d'elles assignades inicialment.
- Graduation List + History File: 16 entrades.
- Cua: 8 entrades.
- Store Buffer: 8 entrades.

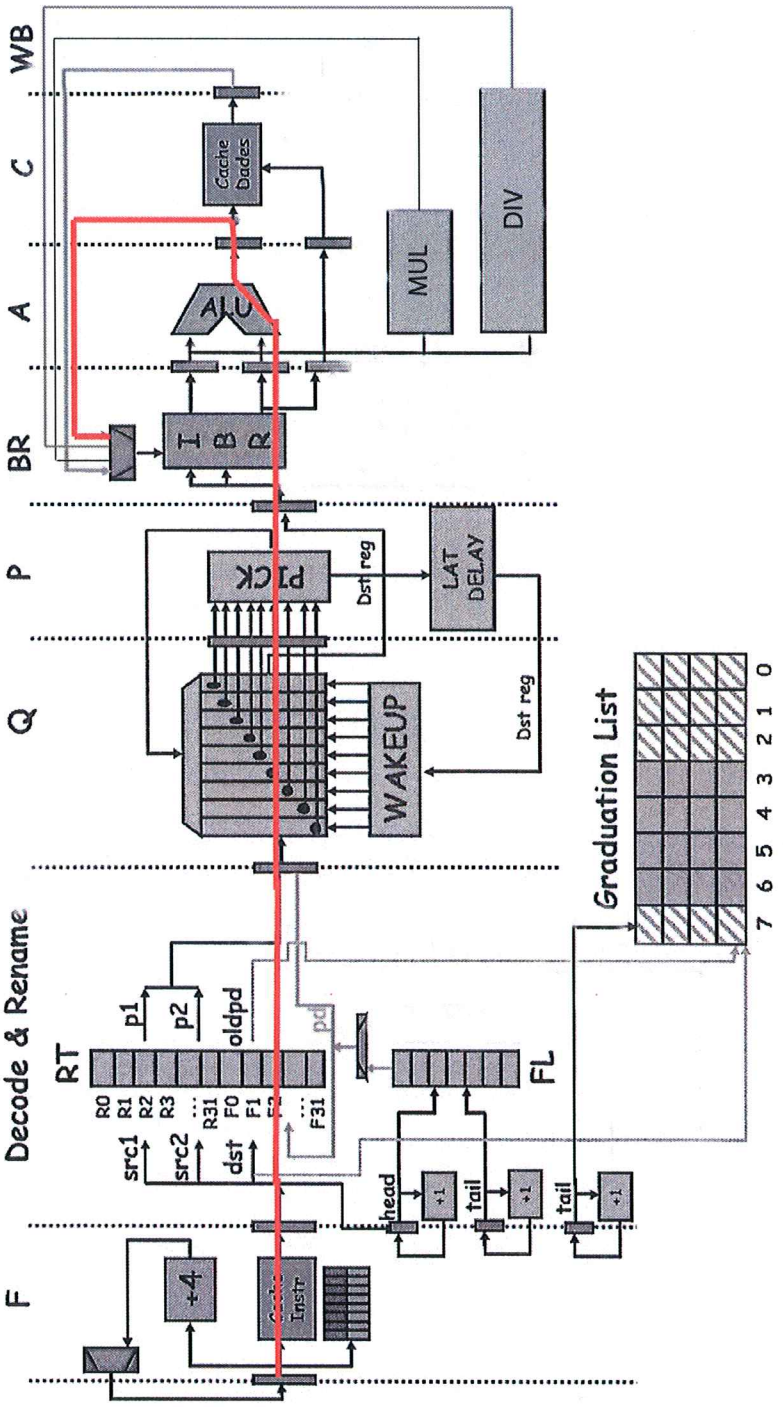

Figura 3.13: Pipeline de RDS

Amb aquests tamanys esperem obtindre un bon compromís entre "guany en rendiment" i "espai", ja que aquestes estructures ocupen una superfície física concreta dins del xip, i no volem desaprofitar espai. Hem escollit els tamanys pensant en aquest factor.