Arquitectura de Computadores

Notas de Clase

Elisardo Antelo Suárez

Tema 4: Núcleos de Procesamento – Paralelismo a Nivel de Instrucción

1- Introdución

Neste tema estudaremos técnicas sofisticadas máis alá da segmentación que permiten acadar un maior paralelismo a nivel de instrución e polo tanto reducir o CPI. Esta diminución do CPI é consecuencia de introducir execución superescalar (é dicir, emisión de varias instrucións en paralelo), execución fora de orde e especulación de control e de datos. Na seguinte sección veremos unha primeira aproximación á execución fora de orde mediante a planificación dinámica baseada en marcador. Logo estudaremos diferentes técnicas que nos permiten especular coa dirección dos saltos. Finalmente amosaremos a estrutura básica dun núcleo actual, con execución superscalar, fora de orde e con especulación dinámica. O tema finaliza cun exemplo concreto de implementación.

2- Planificación Dinámica Baseada en Marcador

O obxectivo é reducir o CPI, facilitando que as instrucións **poidan enviarse á execución fora de orde**. Non obstante, deben respectarse as dependencias do código debidas ao modelo de programación secuencial. Para implementar un pipeline deste tipo necesitamos realizar os seguintes cambios (ver Figura 1):

1- Separar a etapa ID en dúas: Emisión de Instrucións (IE) e lectura de operandos (RR). Na etapa IE decodifícase a instrución e chequéanse os **conflitos estruturais e conflitos WAW** ("Write after a Write"). Na etapa RR a instrución espera a que os datos de entrada á mesma estean dispoñibles (por posibles conflitos de datos, **dependencias RAW**), e cando é o caso, realízase a lectura do banco de rexistros.

2- Incorporamos un búfer de instrucións entre a etapa IF e IE, que permite seguir obtendo novas instrucións da memoria, aínda que non se faga emisión de instrucións. Tamén incorporamos outro búfer entre a etapa IE a a etapa RR, para continuar coa emisión de instrucións aínda que se pare a execución.

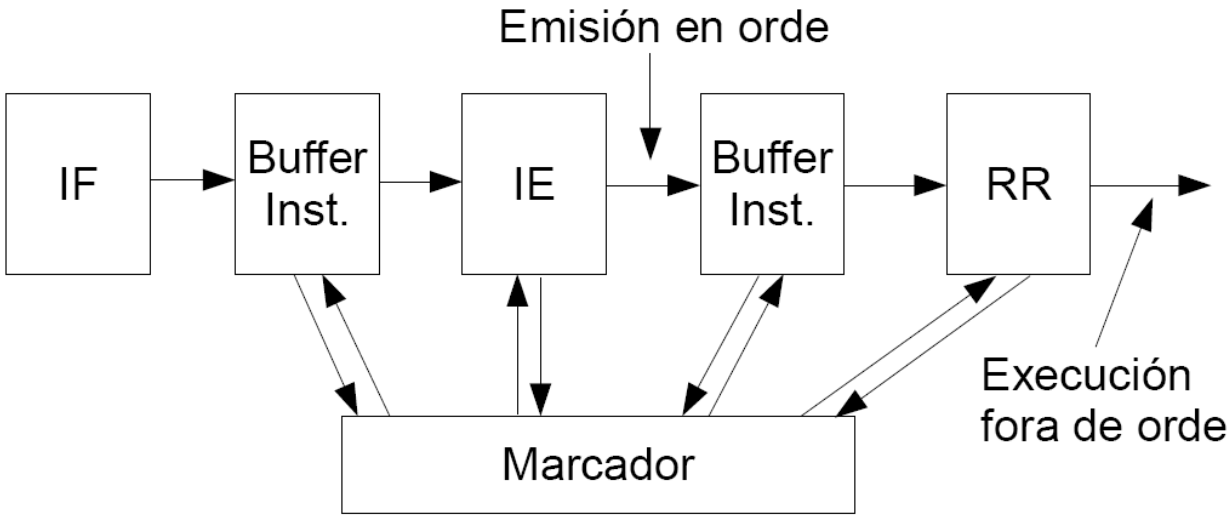


Figura 1: Modificacións no pipeline para planificación dinámica baseada en marcador.

3- Incorporamos un dispositivo de control chamado marcador ("scoreboard" en inglés) para controlar o paso dunha etapa a outra do pipeline. En concreto, realiza os chequeos na etapa IE e RR, para determinar cando unha instrución pode avanzar á etapa de execución. Para a execución pode seleccionarse calquera das instrucións do búfer que cumpran as condicións necesarias para a súa execución (sen conflitos), aínda que sexa **fora da orde orixinal** da secuencia de instrucións.

Debido á posible execución fora de orde, aparecen dous novos conflitos que no aparecían no pipeline básico do tema anterior: conflitos **WAW** (write after a write) e conflitos **WAR** (write after a read, tamén chamado antidependencia). Vexamos un par exemplos para ilustrar estas situacións.

Exemplo de conflito **WAW**:

ADD.D F10, F0, F8

....

SUB.D F10, F5, F4

Estas dúas instrucións teñen o mesmo rexistro destino, e a escritura debe facerse mantendo á orde do programa. A opción utilizada é **parar a emisión** dunha instrución se se detecta que hai xa unha instrución no pipeline que ten como destino o mesmo rexistro. A ocorrencia deste conflito é pouco probable, como xa mencionamos no tema anterior (implementación do pipeline con execución multiciclo), e esta é unha solución cunha implementación moi simple.

Exemplo de conflito **WAR**:

ADD.D F10,F0,F8

DIV.D F0,F2, F10

SUB.D F8,F8,F14

Existe unha antidependencia entre ADD e SUB a través do rexistro F8. Isto implica que SUB non debería realizar a escritura no rexistro antes de que ADD faga a súa lectura.

Además destes dous tipos de conflitos, pode presentarse o conflito **RAW** (read after a write, xa estudado no tema anterior), que constitúe unha dependencia de datos pura. O código exemplo anterior contén unha dependencia **RAW** entre DIV e ADD a través do rexistro F10.

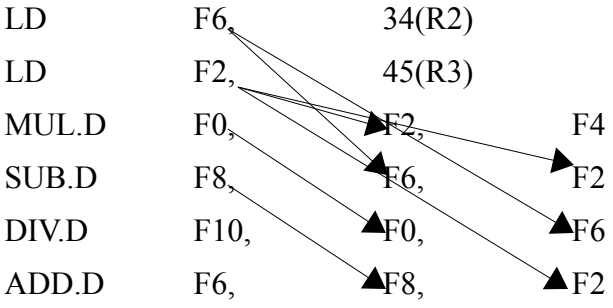
A detección dos diferentes conflitos faise nas seguintes etapas:

- Os conflitos **WAW** detéctanse na etapa **IE**.
- Os conflitos **RAW** detéctanse na etapa **RR**.
- Os conflitos **WAR** detéctanse na etapa **WB**. Se existe posibilidade deste conflito, a instrución para (Stall) na etapa **WB** até que poida escribir o rexistro unha vez solucionado o conflito (cando a intrución coa que ten conflito faga a lectura de rexistros). Consideramos que a etapa **WB** está inmediatamente logo da etapa de execución.

Vexamos un exemplo completo dun código con diferentes tipos de dependencias.

```
LD      F6, 34(R2)
LD      F2, 45(R3)
MUL.D   F0, F2, F4
SUB.D   F8, F6, F2
DIV.D   F10, F0, F6
ADD.D   F6, F8, F2
```

As dependencias **RAW** son as seguintes (indicadas con frechas):



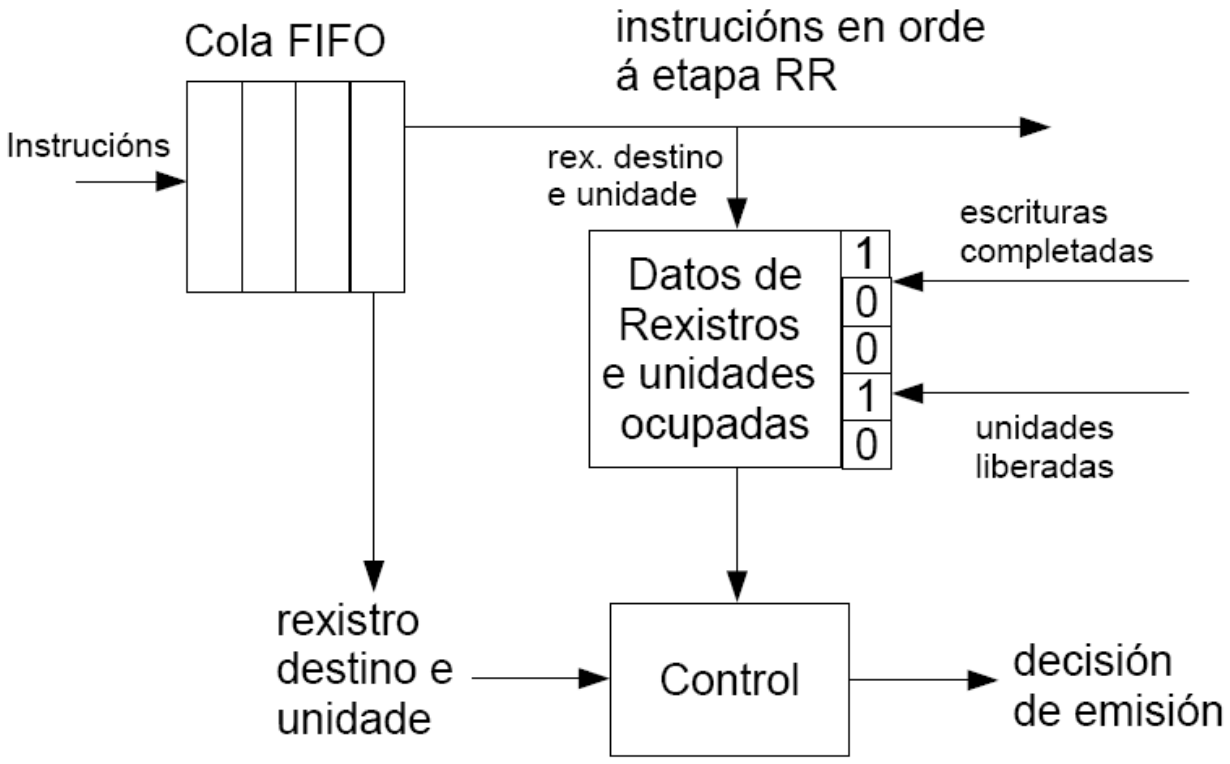


Figura 2: Esquema de alto nivel para o control na etapa IE.

Existe unha dependencia **WAW** entre LD e ADD.D a través do rexistro F6. Existe unha dependencia **WAR** (antidependencia) entre DIV e ADD a través do rexistro F6.

A Figura 2 amosa un diagrama de alto nivel para o control na etapa IE. As instrucións decodifícanse e almacénanse nunha cola FIFO (primeiro en entrar primeiro en saír), de tal xeito que sairán da cola en orde. Cada vez que se emite unha instrución, indícase o número de rexistro que será escrito (almacenado nun vector de elementos binarios), e a unidade que ocupará (tamén un vector binario). Para considerar unha instrución para emisión, determínanse os rexistros que escribirá e a unidade de execución que necesita. O control compara estes valores coa información almacenada sobre as instrucións anteriores e indica se se procede ou non á emisión da instrución.

Das etapas seguintes do pipeline (EX e WB) chegan sinais indicando a escritura de rexistros e liberación de unidades funcionais, para así actualizar a información almacenada que determina se se poden emitir as instrucións.

A Figura 3 amosa o diagrama de alto nivel para o **control na etapa RR**. Cada instrución na cola incorpora información ("flags") relativa ao estado da instrución: en espera, en execución ou rematada. Para cada instrución determínanse as dependencias **RAW** mediante a inspección das instrucións antecedentes na cola (se unha instrución da cola xa rematou, non indicará unha dependencia xa que a escritura de rexistros xa se fixo). Se a instrución non ten dependencias, actívase o sinal "Ready". Un árbitro recibe todos os sinais "Ready" das diferentes instrucións, e selecciona a que se debe executar (isto é simplemente un codificador con prioridade, de tal xeito que se da prioridade ás instrucións con maior antigüidade na cola, para eliminar posible dependencias **RAW** canto antes).

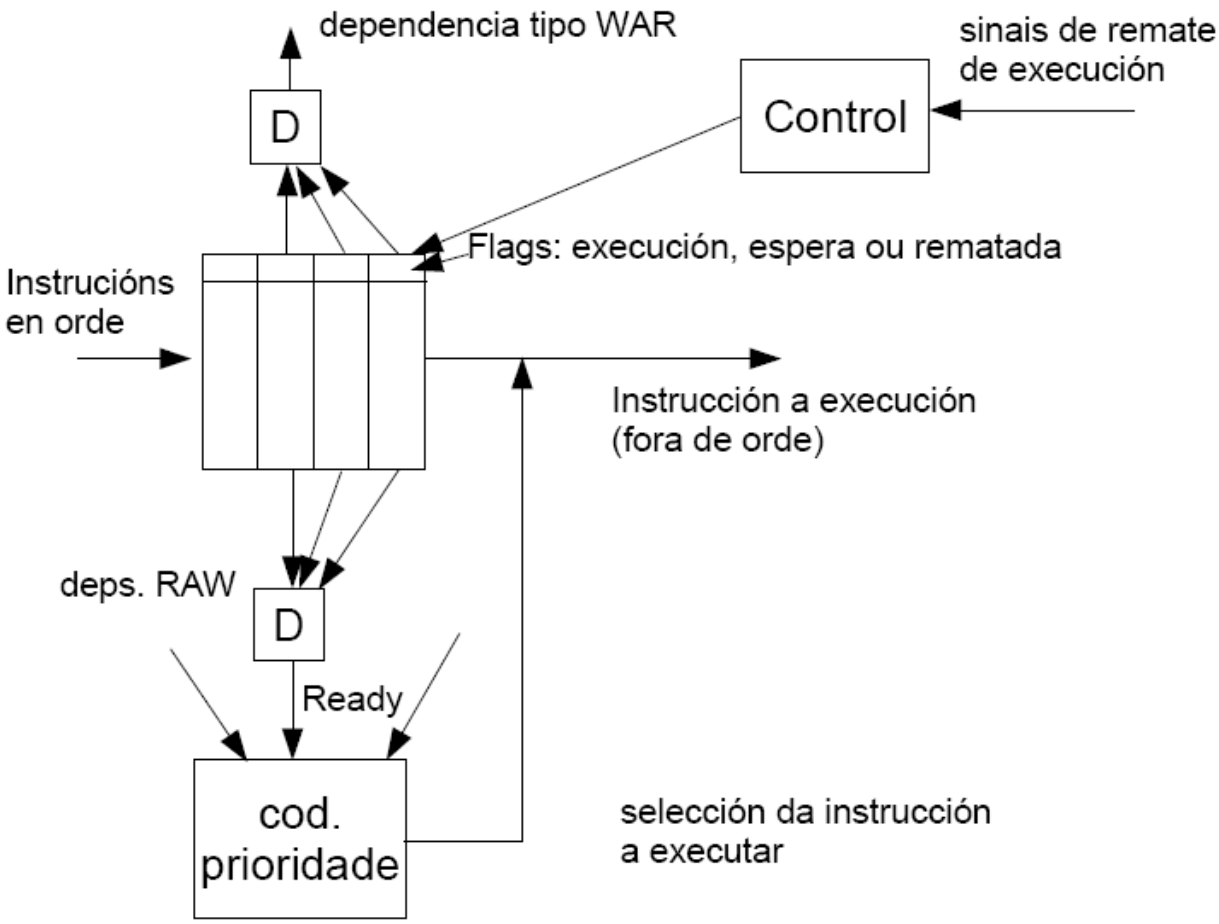


Figura 3: Esquema de alto nivel para o control na etapa RR.

Nesta etapa tamén se determinan as dependencias **WAR** para cada instrución. Se cando se manda a execución unha instrución, o "flag" de dependencia WAR está activado, entón cando a instrución chegue á etapa WB debe comprobarse se o "flag" segue activado, en cuxo caso a instrución non continuará, e espera ("stall") até que o "flag" se desactive.

A pesar da mellora respecto ao pipeline básico, este esquema presenta as seguintes limitacións:

- Non tivemos en conta a posible presencia de **instrucións de salto**. Só analizamos bloques de instrucións consecutivas. Non obstante, os saltos son relativamente frecuentes, co que os tamaño dos bloques sen saltos non ofrecerían moitas posibilidades para aumentar o paralelismo a nivel de instrución.
- A efectividade da planificación dinámica, depende do paralelismo dispoñible entre instrucións (dependencias), da ventá de instrucións consideradas, e do tipo e número de unidades funcionais.
- A planificación dinámica vese limitada por dependencias tipo **WAR** e **WAW**.
- **O estado da máquina e da memoria actualizase fora de orde**, co que non soporta execcións precisas.

Na seguinte sección discutimos algunhas técnicas para predicir a dirección dos saltos, de tal xeito que se poida levar a cabo execución especulativa de instrucións (posteriores á de salto) cun elevado grao de éxito.

3- Predición de Saltos

Como xa vimos no tema anterior, as situacións que implican posibilidade dun salto dan lugar a unha perda potencial de prestacións, xa que pode ter un custo en ciclos se a ruta de instrucións que se segue logo da instrución de salto non é a correcta.

Os procesadores incorporan hardware para predicir a dirección dos saltos e o seu destino. No tema anterior xa estudamos algunhas técnicas básicas baseadas no compilador para reducir o efecto de penalización por saltos. Neste tema revisaremos estratexias dinámicas baseadas en hardware.

Tense observado que os saltos seguen certos patróns regulares (é dicir, non son totalmente aleatorios), que permiten predicir con alta probabilidade o próximo resultado para unha instrución de salto (**tomado ou non tomado**), baseado en información do resultado en ocorrencias anteriores do mesmo salto ou dun conxunto de saltos. O exemplo típico destes patróns son os lazos:

```
for(i=0;i<10;i++){
        corpo do lazo;
}
```

Este código leva implícito unha instrución de salto, no que, se está colocado ao principio do lazo, será non tomado en 10 ocasións, e tomado unha vez. Os principais aspectos a ter en conta no deseño do predictor de saltos son os seguintes:

- Política de cambio de predición en función dos fallos/acertos anteriores.
- Información local: tense en conta a historia temporal do salto que se intenta predicir.
- Información global: tense en conta a historia doutros saltos anteriores, que poden ser o mesmo que o que se está interesado en predicir ou outros diferentes.

3.1 Política de cambio de predición

A técnica básica consiste en almacenar 1 bit, indicando a predición para a seguinte ocorrencia da instrución de salto. O bit non cambia se a predición é correcta, e debe complementarse se é incorrecta. Por exemplo, pódese tomar 0 para predicir non tomar o salto, e 1 para tomar o salto. A Figura 4 amosa un diagrama de estados que ilustra o funcionamento deste esquema.

Un problema importante deste esquema é que aínda que o salto se tome habitualmente, se por exemplo, se produce un fallo de predición (porque non se segue o salto), e no seguinte o salto volve a tomarse habitualmente, entón prodúcense dous fallos de predición:

Predición	1	1	1	1	1	0	1	1	1
Real	1	1	1	1	0	1	1	1	1

Polo tanto, resulta de interese “filtrar” cambios esporádicos no comportamento do salto para quedarse co comportamento máis estable.

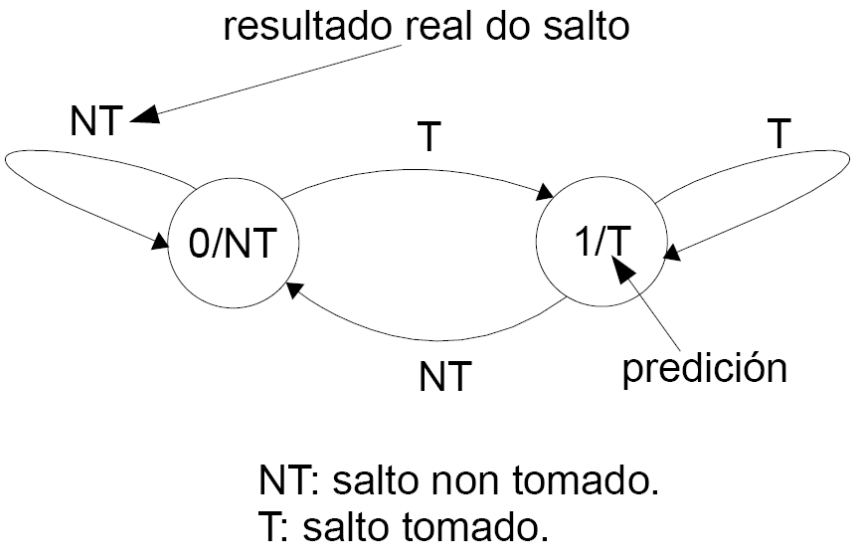


Figura 4: Diagrama de estados para o cambio de predición.

Polo tanto, un esquema que produce mellores resultados, e que se utiliza habitualmente é o de 4 estados, con dous bits para almacenar o estado. Existen diferentes variantes, pero o obxectivo é o mesmo, filtrar cambios esporádicos no comportamento do salto. A Figura 5 amosa o diagrama de estados de dúas variantes que se teñen proposto. Agora existen dúas versións para a predición de salto tomado e salto non tomado, unha é unha predición forte e outra é unha predición feble. Se por exemplo a predición está nun estado forte, e faise unha predición equivocada, pasa a un estado feble, pero segue a facer a mesma predición. Se se fai unha predición equivocada nun estado fleble, entón cambia o sentido da predición (a feble ou forte segundo a variante elixida).

Tamén existen implementacións que utilizan 3 bits de estado, o que permite un maior grao de eficiencia na filtración de comportamentos esporádicos do salto.

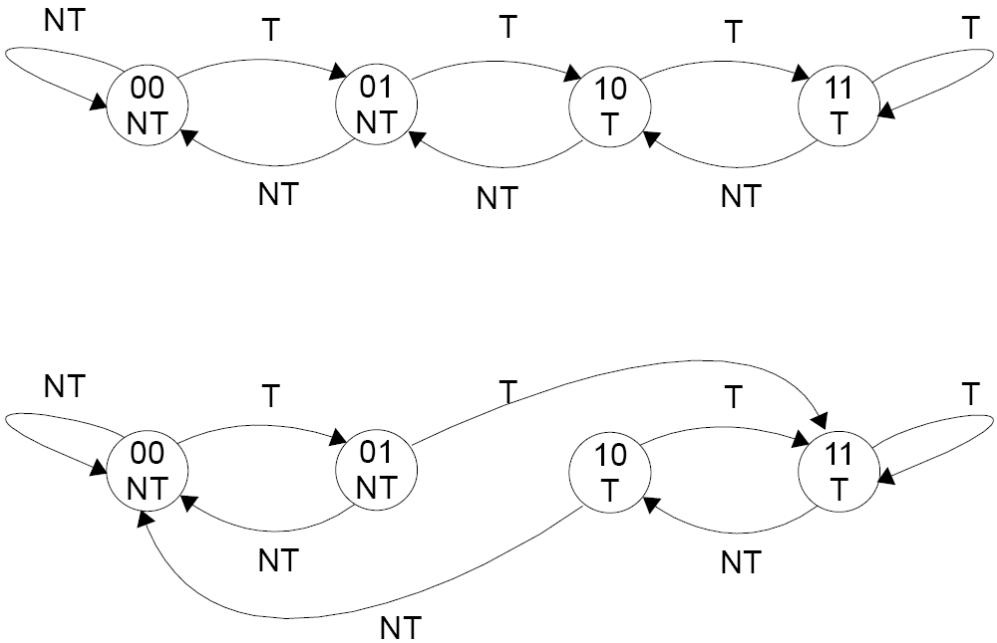


Figura 5: Diagrama de estados para a decisión de dous bits.

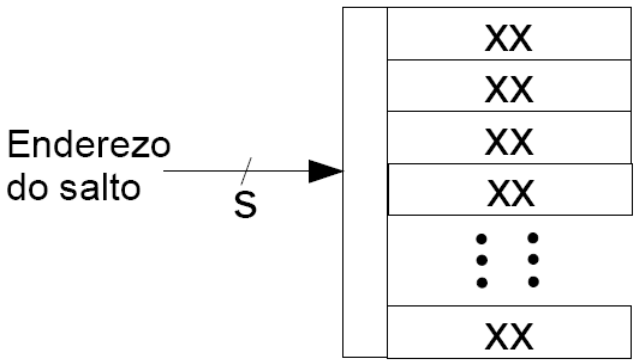


Figura 6: Implementación básica da predición de saltos.

Para a implementación utilízanse os mesmos principios que para memorias cache. Cóllese a parte menos significativa do enderezo da instrución de salto (o número de bits determina o tamaño da cache), e utilízase para extraer o estado de predición almacenado na cache. Cando se resolve o salto, almacénase o novo estado na mesma posición (se hai cambio de estado). Loxicamente, pode ocorrer que unha instrución de salto utilice información doutro salto (cando coinciden os bits menos significativos do enderezo das dúas instrucións de salto). Non obstante, polo principio de localidade, este tipos de situacións teñen baixa probabilidade. A Figura 6 ilustra esta implementación.

3.2 Información Local

Neste caso tense en conta a historia dun mesmo salto para tomar a decisión. Entón almacénase o resultado de cada salto (tomado ou non tomado), correspondente a un certo número de ocorrencias do salto con anterioridade, e con estes bits accedese á información de predición (bits de estado).

A Figura 6 amosa un diagrama de bloques correspondente a esta implementación. Os s bits menos significativos do enderezo da instrución de salto acceden a unha memoria que almacena a historia de k ocorrencias anteriores do salto (ou doutro salto co que coincida nos s bits menos significativos do enderezo, pero xa indicamos antes que isto é pouco probable). Coa historia de k bits do salto accédese a outra memoria que almacena os bits de estado que implementan o filtro de decisión, tal e como comentamos na subsección anterior. Desta táboa obtense a predición. Logo de coñecer o resultado real do salto, é necesario actualizar o contido da memoria de historia do salto, é facer o cambio de estado no filtro de decisión (na posición que se utilizou para facer a predición), accedendo á segunda memoria.

Por exemplo, esta estratexia evita o 50% dos fallos da estrutura básica de predición de saltos (Figura 6) para secuencias de saltos do tipo ...010101010101...Coa estrutura básica o estado oscila entre non tomado forte e non tomado feble (asumindo qu e o estado inicial era non tomado forte), pero a predición é sempre non tomado. Coa estrutura de información local, utilízase dous filtros de decisión diferentes, xa que temos dúas secuencias distintas de historias (por exemplo para 4 bits de historia, estas secuencias son 0101 e 1010). Deste xeito lógrase predicir correctamente o 100% dos saltos.

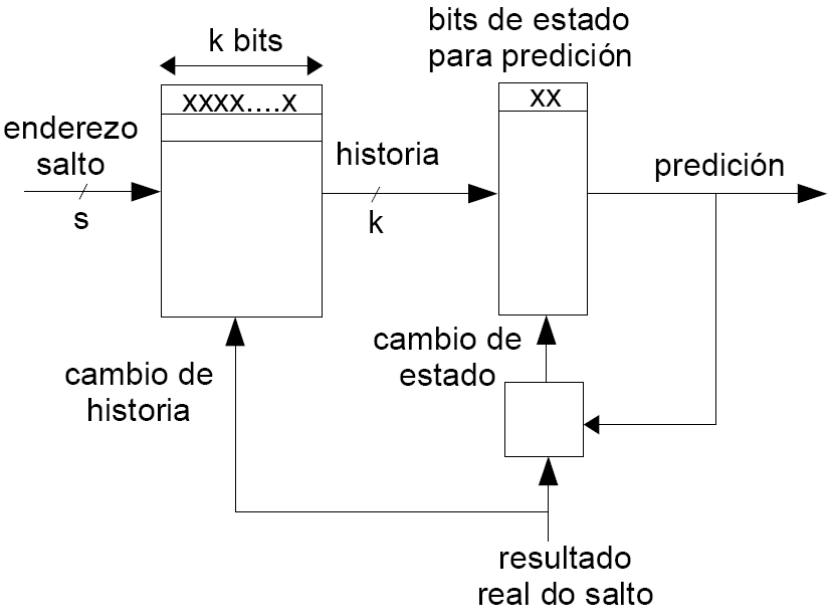


Figura 6: Implementación de alto nivel do predictor local.

3.3 Información Global

Neste caso utilízase a historia de comportamento (tomado ou non tomado) correspondente ás últimas N instrucións de salto. Para manter a historia dos últimos N saltos, é necesario incorporar un rexistro de desprazamento. A Figura 7 amosa a implementación. O resultado do último salto introdúcese pola entrada serie dun rexistro de desprazamento, que contén a historia dos N saltos anteriores (o desprazamento fai que o resultado do salto máis antigo desapareza do rexistro). O contido do rexistro indexa unha memoria que garda os bits de estado correspondente ao filtro de decisión. É posible combinar información local e global, indexando a memoria con bits do enderezo do salto actual e bits correspondentes á historia dos últimos N saltos.

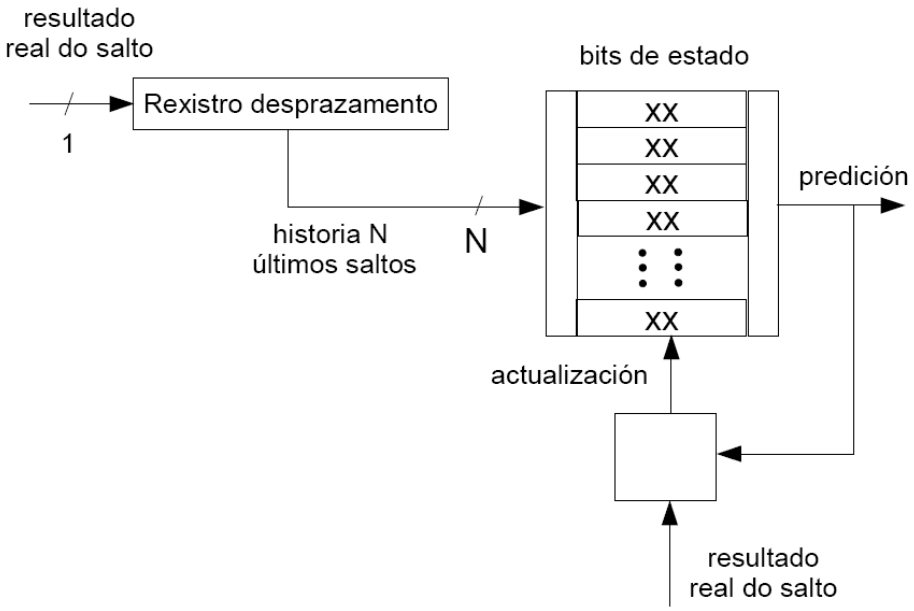


Figura 7: Implementación de alto nivel do predictor global.

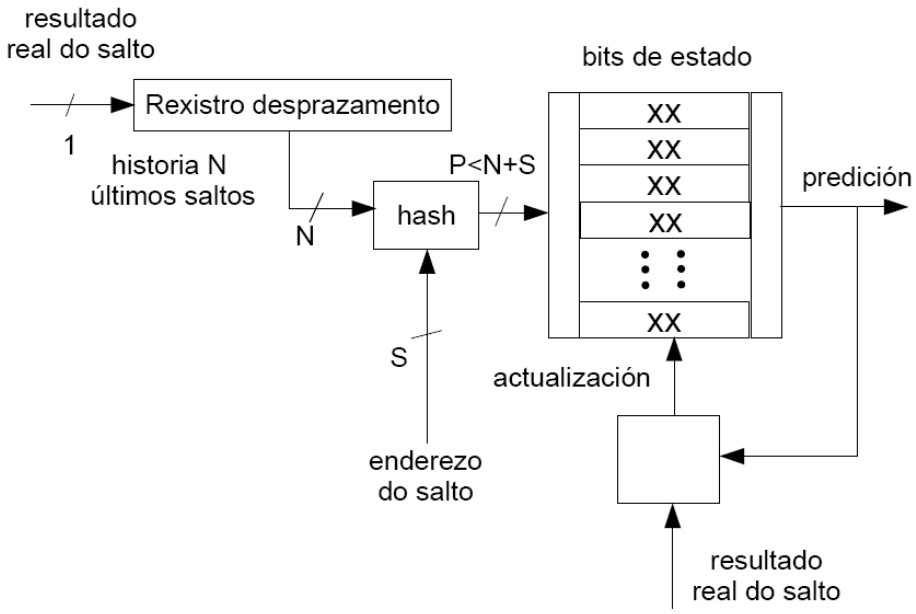


Figura 8: Implementación alternativa do predictor global.

A Figura 8 amosa esta implementación alternativa. A combinación de bits de ambos campos non ten porque ser unha concatenación. Pódese utilizar un hardware que combine os dous conxuntos de bits para producir un conxunto de bits inferior que a suma dos de entrada, para reducir o tamaño da memoria.

3.4 Predictores Combinados

En xeral os procesadores máis avanzados combinan diferentes técnicas que se utilizan de xeito adaptativo ao longo da execución dos programas. Na Figura 9 ilustramos unha destas implementacións. Esta implementación consta dun predictor global cunha memoria de 4K entradas, almacenando 2 bits de estado por entrada. O predictor local ten unha memoria de 1K entradas para almacenar a historia das 10 últimas ocorrencias do salto. Coa historia do salto indéxase unha memoria de 1K entradas, almacenando 3 bits de estado por entrada para implementar un filtro de 8 estados. Para a selección adaptativa do tipo de predictor que se debe utilizar (local ou global), incorpórase unha memoria indexada polos 12 bits menos significativos do enderezo do salto (polo tanto a memoria consta de 4K entradas), que almacena 2 bits de estado correspondentes ao filtro de decisión para tomar o resultado do predictor local ou do global. En total esta implementación específica require 29Kbits de almacenamento. Na actualidade os microprocesadores avanzados utilizan >30Kbits para almacenamento destinado a predición de saltos. Esta cantidade escala coa tecnoloxía, xa que se dispón de máis almacenamento por unidade de área.

En xeral os programas que traballan con enteiros presentan un peor comportamento en canto a predición de saltos, en comparación cos programas con cargas de computación en punto flotante. A taxa de acerto dos saltos depende bastante do tipo de programa, e usualmente atópase entre o 85-99%.

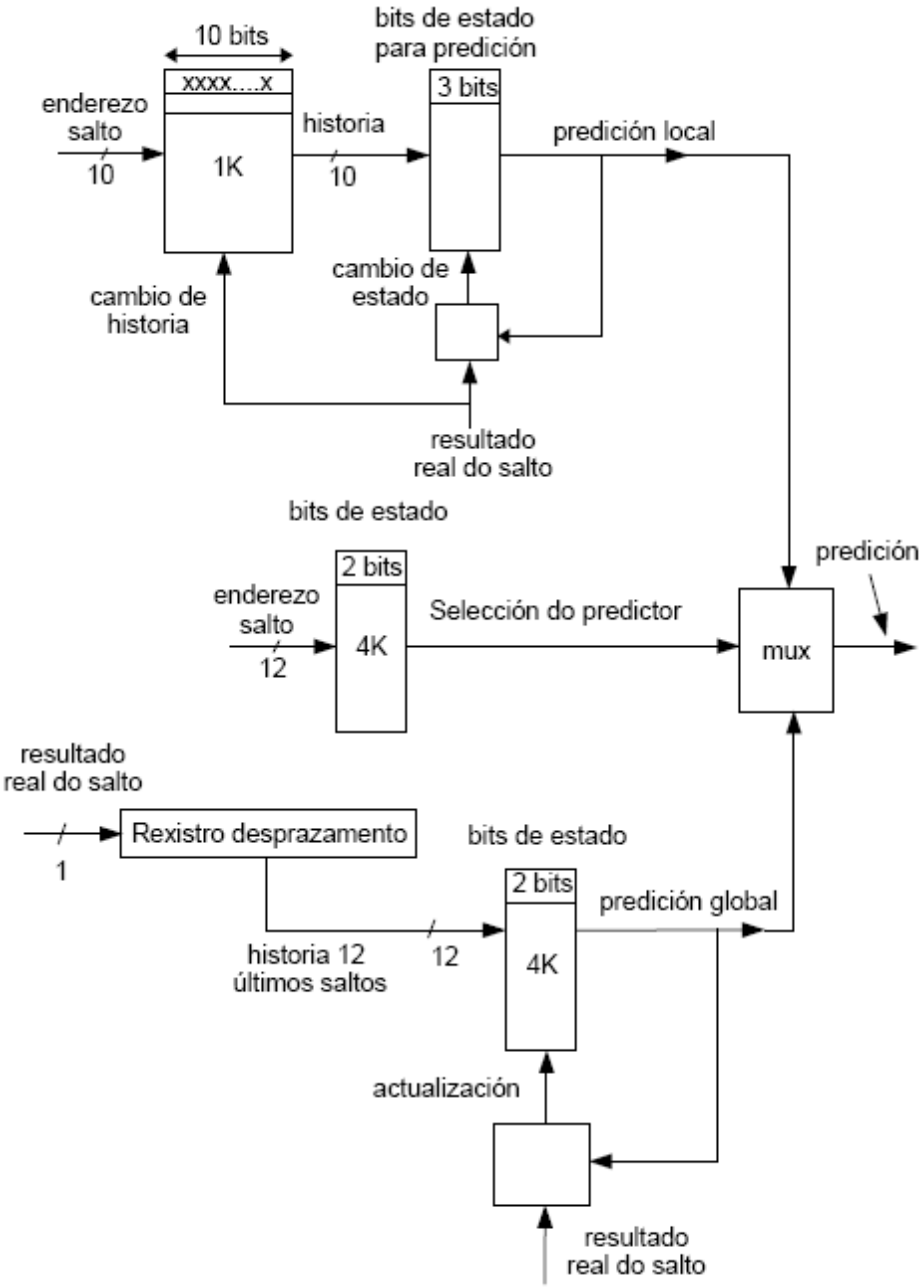


Figura 9: Unha implementación particular do predictor combinado.

O impacto na latencia que teñen os saltos dun programa, dependen polo tanto de:

- Frecuencia das instrucións de salto (é dicir, a porcentaxe de instrucións de salto).
- Custe da mala predición dos saltos.
- Taxa de acerto da predición de saltos.

Por exemplo, se o 20% son instrucións de salto, cun acerto de predición do 95%, e cun custe de predición errónea de 10 ciclos, a **influenza resultante no CPI** é de 0.2 saltos/instrución x 0.05 fallos/salto x 10 ciclos/fallo=0.1 ciclos/instrución.

3.5 Predictores Específicos para Lazos

En algúns procesadores incorpórase unha estrutura específica para detectar e predicir o comportamento dos saltos asociados a lazos.

Os lazos presentan un só cambio na dirección do salto, e van na mesma dirección de salto durante todas as iteracións do lazo. Por exemplo, o lazo

```
for(i=0;i<8;i++) {
        .....
}
```

ten o seguinte patrón de salto (supoñendo salto ao principio do lazo, e 1 tomado 0 non tomado): 000000001.

Esta estrutura é basicamente un contador de iteracións para detectar o período do lazo. A información que debe gardarse é se o salto (o enderezo da instrución de salto) ten comportamento de lazo ou non, se o salto se toma ou non á saída do lazo, o período, o número de veces que se leva repetido o lazo na execución actual e o destino do salto ("branch target" en inglés). Toda esta información cóllese en sucesivas execucións do lazo completo. Por exemplo, a familia de procesadores Core 2 de Intel ten un contador de 6 bits, polo que permite predicir lazos cun período máximo de 64 iterations.

3.6 Predición do Enderezo de Destino do Salto

Ademais de predicir se o salto se toma ou non, tamén resulta útil determinar canto antes o enderezo da instrución destino se o salto se toma. Deste xeito, non se terían (con elevada probabilidade) penalizacións polo salto e o pipeline non tería que parar. Por outra banda, nos mecanismos de predición de saltos estudados anteriormente, utilizábase o enderezo da instrución de salto, **pero non se indicaba como distinguir canto antes se a instrución é de salto ou non (ou se no conxunto de instrucións hai polo menos unha instrución de salto se o procesador é superscalar)**. Facer unha predición de salto tomado para unha instrución que non é de salto, resulta obviamente en altas ineficiencias.

O predictor do enderezo de destino do salto (Branch Target Buffer en inglés – **BTB**), é unha cache indexada pola parte menos significativa do enderezo de salto, e donde as etiquetas (tags en inglés) son a parte máis significativa do devandito enderezo (a cache pode ser totalmente asociativa – moi custoso, ou asociativa por conxuntos). O contido asociado a cada entrada é o enderezo do destino do salto correspondente. A Figura 10 ilustra a estrutura de alto nivel do **BTB**. **Só se introducen na táboa os saltos que se toman**. Lóxicamente os datos da táboa serán utilizados só cando o salto actual se predí coma tomado. Nalgunhas implementacións, tamén se garda, ademais do enderezo, a propia instrución. Deste xeito o BTB pode ser máis lento (a instrución de destino non ten que buscarse na cache) e polo tanto pode utilizarse un BTB máis grande con maior rendemento. Se o enderezo resulta nun acerto no BTB, quere dicir que a instrución é de salto (ou conten polo menos un salto para un conxunto de instrucións en caso de execución superscalar). Nese caso, utilizarase adicionalmente o mecanismo de predición para determinar cal debe ser a seguinte instrución a ler da cache de instrucións.

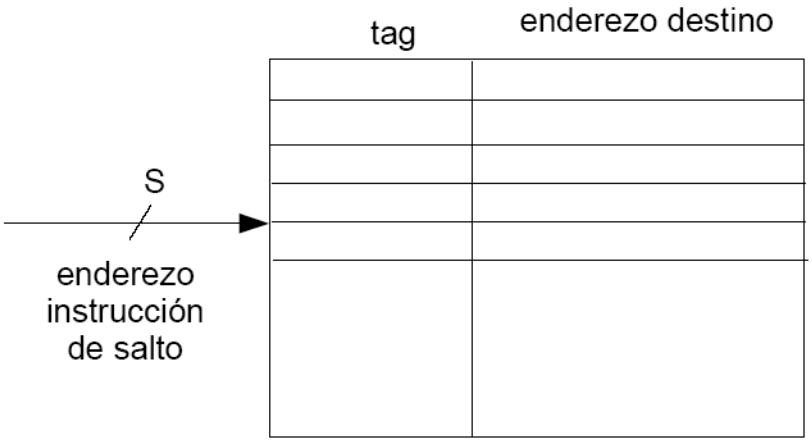


Figura 10: Esquema de alto nivel do BTB.

A Figura 11 amosa a secuencia de eventos no pipeline simple de 5 etapas estudado no tema anterior, cando se incorpora un BTB para predicir o enderezo do salto. A actualización do BTB, logo de coñecerse o resultado real do salto, farase tendo en conta cal sería o resultado da predición na seguinte ocorrencia. Se o resultado da predición é non tomado, entón ese enderezo non figurará no BTB (invalidarase no caso que xa estivera). Se o resultado da predición é tomado, entón debe introducirse no BTB, utilizando o enderezo de destino que se obtivo na execución actual. Existe unha penalización en ciclos se: i) o enderezo do salto está no BTB, e se predí salto tomado, cando o salto realmente non se toma, ii) se o enderezo do salto non está no BTB e o salto se toma (hai que esperar polo cálculo do enderezo de destino do salto).

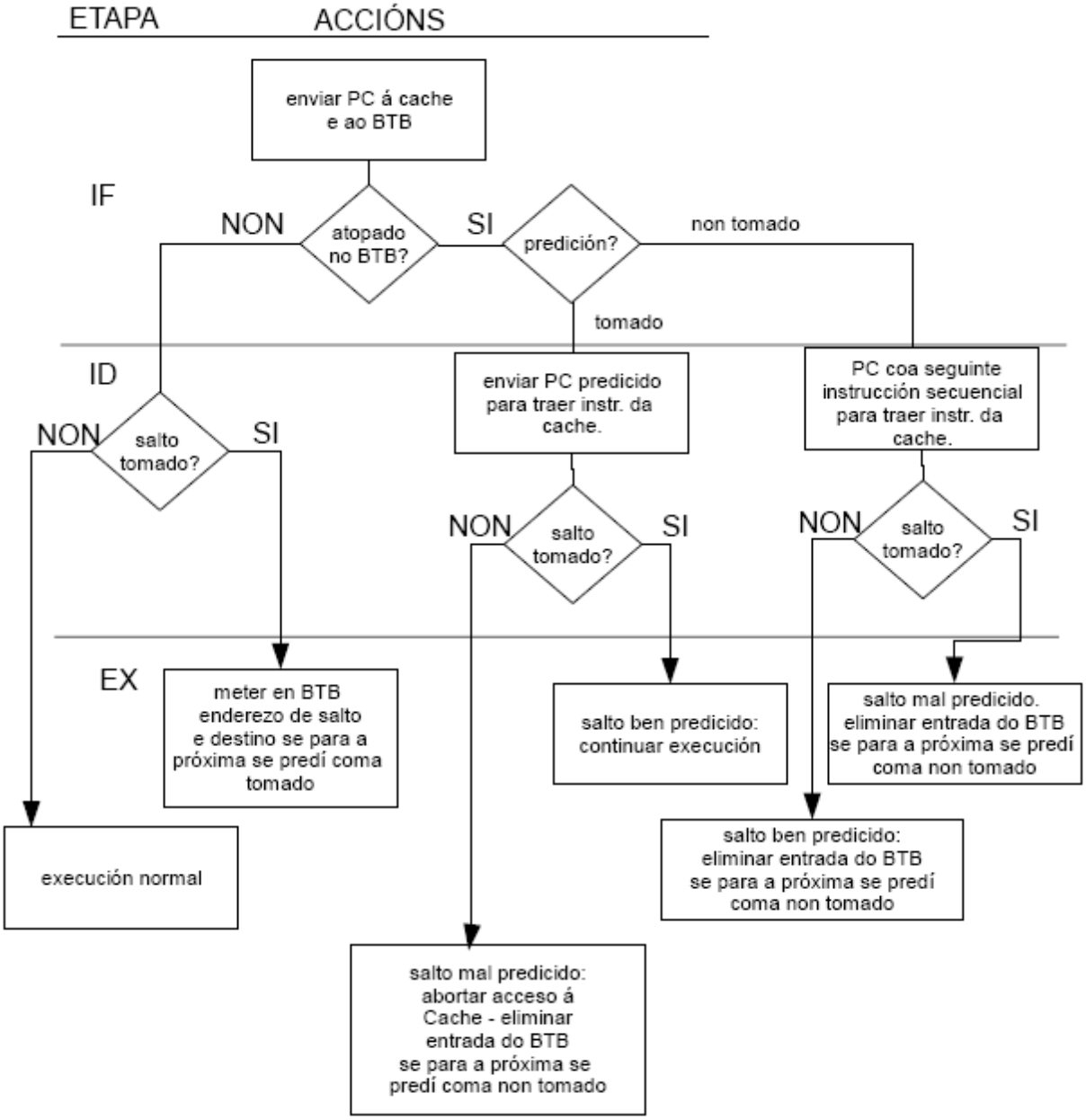


Figura 11: Secuencia e eventos no pipeline incorporando o BTB.

3.7 Predición de enderezo de retorno de chamadas

A chamadas as funcións son un evento con elevada probabilidade dentro do código. Unha chamada a unha función leva implícito un salto incondicional á función, e logo de facer os cálculos da función, un salto incondicional á seguinte instrución dende a que se chamou a función. O problema fundamental é a determinación rápida do enderezo de retorno da chamada, para poder seguir co fluxo de instrucións do pipeline, sen paradas. Para acadar isto, incorpórase un búfer en forma de stack (último en entrar, primeiro en saír, ou LIFO). Cada vez que se fai unha chamada a función, almacénase no stack o enderezo da seguinte instrucións secuencial posterior á chamada. Se dentro da función se chama a outra, entón gardarase tamén o enderezo de retorno correspondente. Logo de que a función máis interna remata, ao executarse unha instrución de retorno de chamada, o enderezo de retorno obtense do stack, en orde inversa. A medida que se executan as diferentes instrucións de retorno, os enderezos son lidos do stack (e eliminados).

4 Arquitectura dun Núcleo con Paralelismo a Nivel de Instrución

Nesta sección presentamos os elementos fundamentais da arquitectura dun núcleo que explota de xeito agresivo o palalelismo a nivel de instrución para reducir o CPI. As características fundamentais que explotan estes núcleos son as seguintes:

- Emisión e execución de instrucións de xeito superscalar: para diminuír o CPI é necesario executar varias instrución en paralelo. Para isto engádense varias unidades funcionais de execución, e é necesario incorporar mecanismos para emitir, decodificar e despachar varias instrucións por ciclo para manter as unidades funcionais ocupadas. Este modelo é coñecido coma execución superscalar (isto xa se podería ter incorporado aos pipelines sinxelos que levamos estudado, pero con pouca eficiencia).
- Execución fora de orde: as dependencias de datos (tipo RAW), limitan o número de instrucións que podemos atopar en orde para lanzar a execución en paralelo. Polo tanto un mecanismo natural para aumentar o paralelismo a nivel de instrución, é o de seleccionar instrucións para ser executadas, fóra da orde do programa orixinal, evitando a limitación das dependencias de datos coa execución en orde.
- Especulación: as dependencias de control (saltos condicionais) son unha seria limitación para acadar un elevado paralelismo a nivel de instrución, como xa mencionamos anteriormente. Como vimos, os saltos poden predicirse cunha elevada probabilidade, co que resulta interesante especular cal será a seguinte instrución logo da instrución de salto, sen ter que esperar un certo número de ciclos a que se resolva a dirección que toma o salto e, se é necesario, o enderezo de destino. Obviamente, é necesario incorporar algún mecanismo que permita manter a execución correcta, no caso (pouco probable) de que a especulación fose incorrecta. Con esta mesma idea, tamén é posible realizar especulación de datos: por exemplo especular lecturas de datos (que poden depender dunha instrución de escritura anterior que aínda non se completou).
- Excepcións e interrupcións precisas e semántica do programa secuencial: malia que o núcleo poida executar as instrucións fora de orde, a alteración definitiva do estado do núcleo (rexistros visibles a nivel do repertorio de instrucións) ou da memoria, ten que respectar a orde orixinal do programa secuencial (tal e como o entende o programador). Isto é especialmente necesario, tendo en conta que poden ocorrer excepcións e interrupcións durante a execución, polo que o estado do núcleo e da memoria ten que ser consistente coa semántica do programa que se está a executar (por exemplo para permitir reanudar a execución dende a instrución que produciu unha excepción).

Nas seguintes subseccións, vamos a describir os elementos fundamentais que compoñen a arquitectura dun núcleo deste tipo. Veremos cal é a función de cada bloque e súa interacción co resto en conexión coas características que acabamos de enunciar.

4.1 Renomeado de Rexistros

A arquitectura dun repertorio de instrucións (ISA en inglés – Instruction Set Architecture) constitúe a interface fundamental entre o hardware e o compilador (software). Unha ISA describe unha serie de rexistros hardware que serán os que utilice o compilador para almacenar os operandos que están en memoria, para operar con eles de xeito temporal e obter resultados (que se almacenarán temporalmente neses rexistros). Chamaremos a estes, rexistros ISA, ou rexistros arquitecturales. Non obstante unha ISA pode ter diferentes implementacións, con máis ou menos complexidade. Por exemplo, o escalamento da tecnoloxía permite utilizar máis recursos hardware de xeración en xeración. Polo tanto é posible que unha implementación específica conte cun maior número de

rexistros, que os rexistros ISA. Para utilizar estes rexistros extra, introdúcese a técnica de renomeado de rexistros.

O renomeado de rexistros, consiste nunha función de correspondencia temporal e dinámica, entre os rexistros ISA e o resto de rexistros que ten a implementación. Deste xeito logo de decodificar cada instrución, o rexistro ISA que se especifica como receptor do resultado é renomeado a un dos rexistros non ISA dispoñibles. Mentres o rexistro ISA permaneza renomeado, calquera referencia de lectura dunha instrución posterior ao devandito rexistro, utilizará o rexistro non ISA. O renomeado finaliza cando o mesmo rexistro ISA é o receptor dun resultado noutra instrución (faise un novo renomeado dese rexistro), ou se antes remata definitivamente a execución da instrución que produciu o renomeado. En calquera caso, cando remata definitivamente a instrución, realizarase unha copia do resultado do rexistro non ISA ao rexistro ISA, para que o estado da máquina sexa consistente co programa en execución (mantendo así excepcións e interrupcións precisas).

Este sistema presenta dúas vantaxes claras: i) elimínanse as dependencias tipo WAW e WAR e ii) facilita as técnicas de especulación, ao ter os valores especulativos en rexistros non ISA. Cando a condición especulativa remata, se os datos son válidos, poden copiarse nos rexistros ISA.

Por exemplo, se os rexistros ISA de punto flotante son F0, F1, F2,..., e existe un conxunto adicional de rexistros RF0, RF1, RF2,....., é posible romper as dependencia WAW e WAR no seguinte código:

```
DIV.D F0,F2, F4
ADD.D F10,F0,F8
SUB.D F8,F8,F14
ADD.D F0, F3, F5
```

Existe unha dependencia WAW a través de F0 entre DIV e o último ADD. Existe unha dependencia WAR a través de F8, entre o primeiro ADD e SUB, e outra a través de F0 entre o primeiro ADD e o último. Supoñamos que F0 se renomea a RF5 logo de decodificar DIV, e a RF8 logo de decodificar o último ADD. Supoñamos tamén que logo de decodificar SUB, F8 é renomeado a RF4. Por último, logo de decodificar o primeiro ADD renomeamos F10 a RF3 (este non é significativo neste exemplo). Logo do renomeado, a execución podería levarse a cabo seguindo o seguinte código:

```
DIV.D RF5, F2, F4
ADD.D RF3, RF5, F8
SUB.D RF4, F8, F14
ADD.D RF8, F3, F5
```

Deste xeito, desaparecen as dependencias WAW e WAR, o que facilita a execución das instrucións fora de orde. Só é necesario ter en conta as dependencias puras, as tipo RAW (entre DIV e ADD a través de RF5).

Unha posible implementación consiste en ter unha táboa para almacenar o renomeado dos rexistros, un banco de rexistros ISA (AR), e un banco de rexistros non ISA (RR). A Figura 12 ilustra esta

Figura 12: Esquema básico do renomeado de rexistros.

implementación. Na táboa de renomeado figura unha entrada para cada rexistro ISA, e almacena o identificador do rexistro no que se renomeou (se é que o fixo). O banco AR consta dunha entrada por cada rexistro ISA, con espazo para almacenar o valor de cada rexistro, un campo B que indica se o rexistro está renomeado, e un campo T, que indica o indentificador do rexistro no que se renomeou. O banco RR constan dunha entrada por cada rexistro non ISA, con espazo para almacenar o valor de cada rexistro, un campo V para indicar se o resultado que almacena é válido (é dicir se a unidade de execución correspondente xa escribiu o resultado da instrución neste rexistro), un campo B que indica se o rexistro corresponde a un renomeamento dun rexistro ISA, e un campo T que indica o rexistro ISA que renomeou.

A Figura 13 ilustra o proceso de lectura de rexistros cando se decodifica unha instrución, tendo en conta o mecanismo de renomeado de rexistros. Co idendificador do rexistro ISA na instrucción, accédese a correspondente entrada do banco AR. Se o sinal B non está activo, entón farase a lectura do contido do rexistro ISA. Se o sinal B está activo implica que o rexistro foi renomeado. Nese caso debe lerse o rexistro renomeado. Sen embargo isto dependerá do campo V (válido) do rexistro correspondente no banco RR. Se o sinal indica que o valor almacenado é válido, faráse a lectura do seu contido. Pola contra, se o campo V indica que o valor non é válido (está pendente de actualizarse por unha instrución en execución), o que se le é o identificador do rexistro (campo T). Este identificador será utilizado posteriormente para conseguir que a instrución que escriba o resultado, faga un bypassing directo a esta instrución.

A Figura 14 ilustra o proceso de escritura de rexistros. Logo da decodificación da instrución, ao coñecer o rexistro ISA destino da instrución, será necesario renomealo. Para isto chequéase no banco de rexistros RR, os que teñen o campo B (ocupado) inactivo. Logo de atopar un rexistro libre, introdúcese o seu identificador na táboa que rexistra o renomeado. Logo da execución da instrución, a unidade de execución fará a escritura do resultado no correspondente rexistro do banco RR (no que se renomeou ao rexistro ISA). Posteriormente, cando proceda (por exemplo, se a instrución remata definitivamente), actualizarase o contido do rexistro ISA no banco AR, e

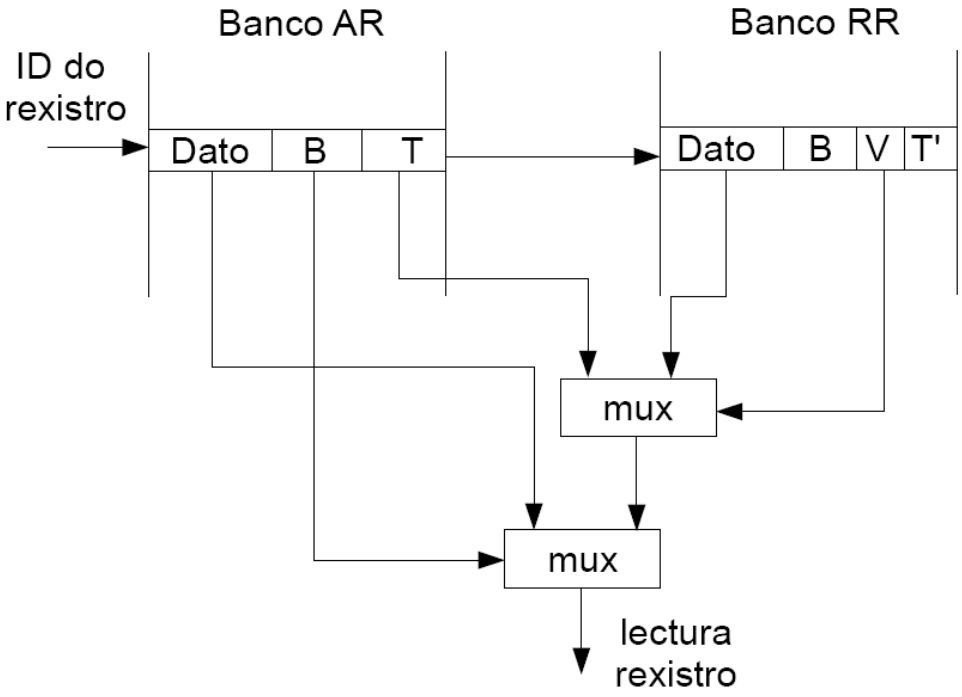


Figura 13: Lectura de rexistros co mecanismo de renomeado.

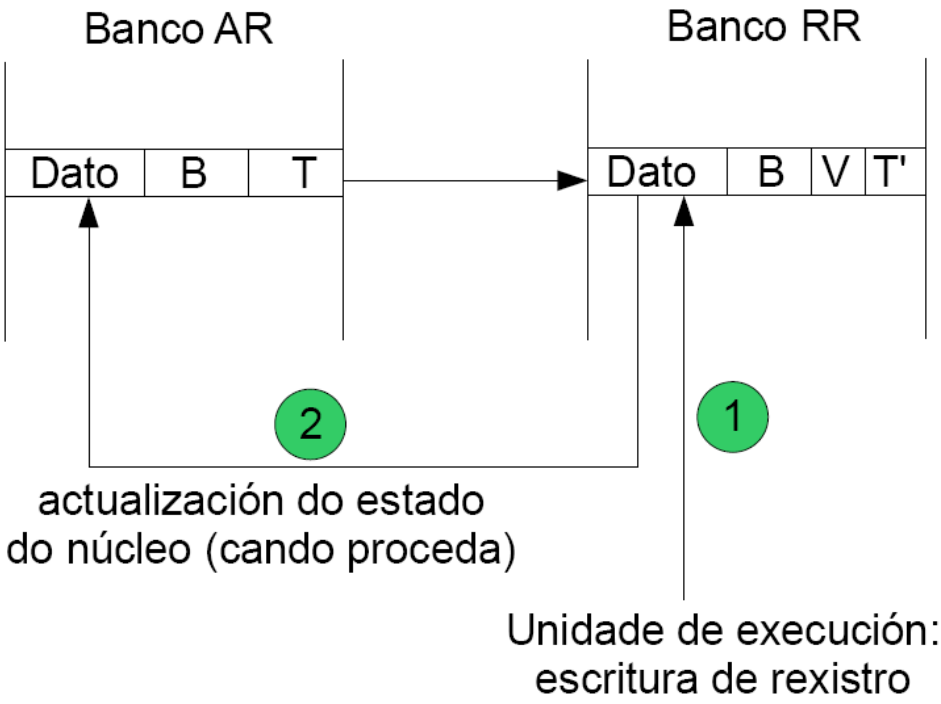


Figura 14: Escritura de rexistros co mecanismo de renomeado.

liberarase o renomeado deste rexistro (se é necesario). Para execución superscalar (emisión de varias instrucións por ciclo), os bancos de rexistros deben dispoñer de varios portos de lectura/escritura, e este número debe escalar co número de instrucións decodificadas en paralelo.

4.2 Estacións de Reserva

As estacións de reserva son unidades para o almacenamento de instrucións en espera para a súa execución. Algúns dos operandos de entrada das instrucións que se almacenan poden non estar dispoñibles inmediatamente, xa que serán producidos por instrucións que están en execución, ou que están tamén en espera. As estacións de reserva poden ter unha arquitectura centralizada para varias (ou todas) as unidades de execución, ou descentralizada (unha para cada unidade de execución, ou pequenos grupos). O sistema centralizado é máis flexible en canto a utilización do almacenamento, pero resulta nunha estrutura con bastante complexidade. As arquitecturas non centralizadas poden incorrer nalgunhas ineficiencias, xa que cada estación de reserva só pode almacenar instrucións dunha unidade de execución, podendo darse o caso de estar unha estación de reserva chea, e polo tanto sen posibilidade de almacenar máis instrucións, cando outras estacións de reserva teñen espazo libre.

A entrada de instrucións nas estacións de reserva é en orde, pero a execución pode ser fora de orde, dependendo da dispoñibilidade dos operandos de entrada das instrucións.

A Figura 15 ilustra un esquema de alto nivel dunha estación de reserva. Consta de varias entradas, unha para cada instrución que será emitida a execución. A estación ten unha unidade de reserva, que recibe peticións da etapa anterior (decodificación e lectura de rexistros) para reservar unha entrada a unha nova instrución (**despacho de instrucións**). Por outra banda conta cun hardware para o control de **emisión** á correspondente unidade de execución (a elixir entre as instrucións listas para emisión). Cada entrada da estación de reserva ten diferentes campos (ver Figura 16): campo B para indicar que a entrada está ocupada por unha instrución, campo I co código da instrución, campo OP1 e OP2 cos operandos de entrada da instrución, e os seus correspondentes campos V1 e V2 para indicar cando os operandos son válidos, un campo R, que indica se a instrución está lista para emisión. As entradas a OP1 e OP2 proceden dos portos de despacho de instrucións (lectura de rexistros) e tamén dos camiños de bypass das unidades de execución. Pode ocorrer que algún dos operandos OP1 ou OP2 sexa un identificador dun rexistro renomeado, indicado que o seu valor está sendo calculado por unha unidade de execución (o bit V correspondente indicará que o operando aínda non é válido). Cando a unidade de execución correspondente, produce o resultado con destino ao rexistro renomeado, os camiños de bypass permiten introducir este valor directamente nas entradas da estación de reserva que teñan o identificador do rexistro renomeado como operando de entrada. Polo tanto para cada entrada, e para cada operando, son necesarios comparadores para identificar cando o contido dun rexistro renomeado está xa dispoñible (nos camiños de bypass das unidades de execución).

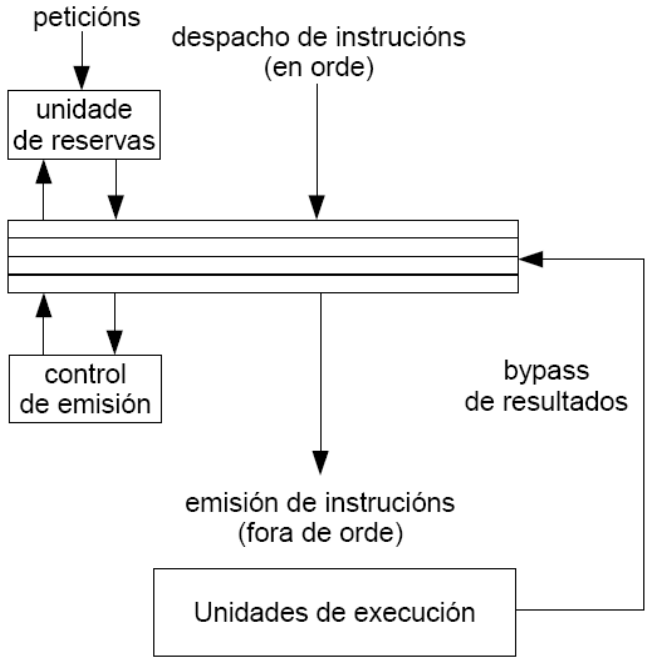


Figura 15: Esquema de alto nivel dunha estación de reserva.

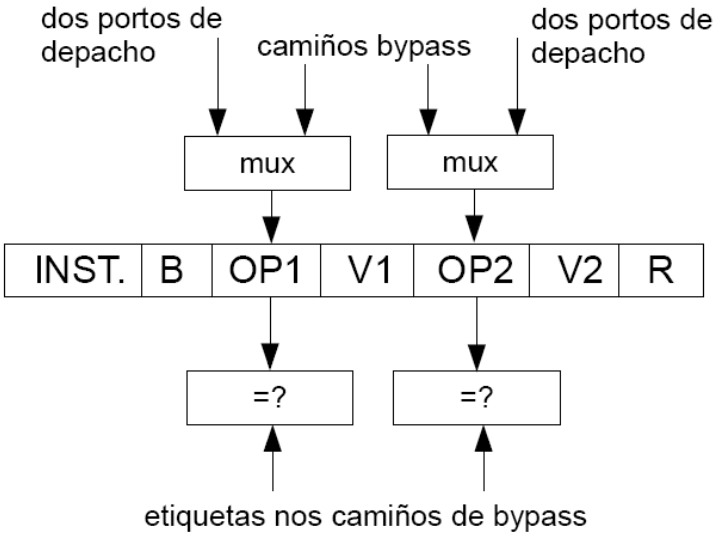


Figura 16: Campos asociados a cada entrada da estación de reserva.

4.3 Búfer de Reordenamento de Instrucións (Reorder Buffer en inglés)

Esta estrutura de almacenamento contén instrucións e información sobre a súa execución, daquelas que xa se despacharon (que posiblemente están nas estacións de reserva, en execución, ou rematadas), pero que aínda non actualizaron o estado da máquina (rexistros ISA) ou da memoria (de xeito xa irreversible). Esta estrutura permite actualizar o estado da CPU e da memoria en orde, de acordo coa semántica secuencial do programa que se executa.

As instrucións no ROB (Reorder Buffer) poden estar nalgún dos seguintes estados:

- En execución: cando a instrución aínda non rematou a execución.
- Finalizado: cando finaliza a execución e produce un resultado que se escribe nun rexistro renomeado (se produce resultado), ou nun buffer se se trata dunha instrución de escritura en memoria (Store), previo á escritura en memoria.
- Completada: cando a instrución actualiza o estado do núcleo, é dicir escribe nos rexistros ISA.
- Retirada: para operacións con resultados con destino a rexistro, completada e retirada é o mesmo. Para operacións de escritura en memoria (Store), a instrución retírase cando se procede á escritura da memoria na cache de datos.

O ROB contén as instrucións dende que se despachan até que son completadas ou retiradas. Para garantir a semántica do programa secuencial, as instrucións son completadas/retiradas segundo a orde orixinal do programa. Polo tanto o ROB incorpora dous punteiros, un sempre apunta á entrada máis antiga e o outro á entrada máis recente (a entrada das instrucións no ROB é en orde), de tal xeito que se coñece a orden na que deben completarse/retirarse. Cando unha instrución se completa/ retira, libera a entrada correspondente no ROB e o punteiro pasa a apuntar a seguinte máis antiga.

O ROB facilita a **especulación** (por exemplo por dependencias de control: saltos), de tal xeito que se incorpora información ás entradas para indicar as instrucións que están en estado especulativo. Unha entrada especulativa non pode ser retirada/completada até que pase a ser non especulativa (por exemplo cando se resolve o salto). Se o resultado da acción especulativa foi negativo, invalídanse as entradas do ROB correspondentes as instrucións especulativas dependentes desa acción (co cal nunca serán completadas/retiradas, e non producirán cambios no estado da máquina).

Existen variantes que combinan o ROB co banco de rexistros renomeados, é dicir, cada entrada do ROB ten sitio para almacenar o resultado logo da finalización da execución.

A Figura 17 amosa a estrutura interna do ROB, indicando os campos que almacena cada unha das entradas. A Figura 18 amosa o esquema de interaccións con outros bloques. O ROB recibe información das estacións de reserva para actualizar o estado das instrucións, e tamén información das unidades de execución (resultados pendentes). En canto á especulación, debe recibir información de si a acción especulativa tivo éxito ou non. É necesario que reciba información do enderezo da instrución a partir do cal as instrucións son especulativas, para pasalas a estado non especulativo se a especulación tivo éxito ou para invalidalas en caso contrario.

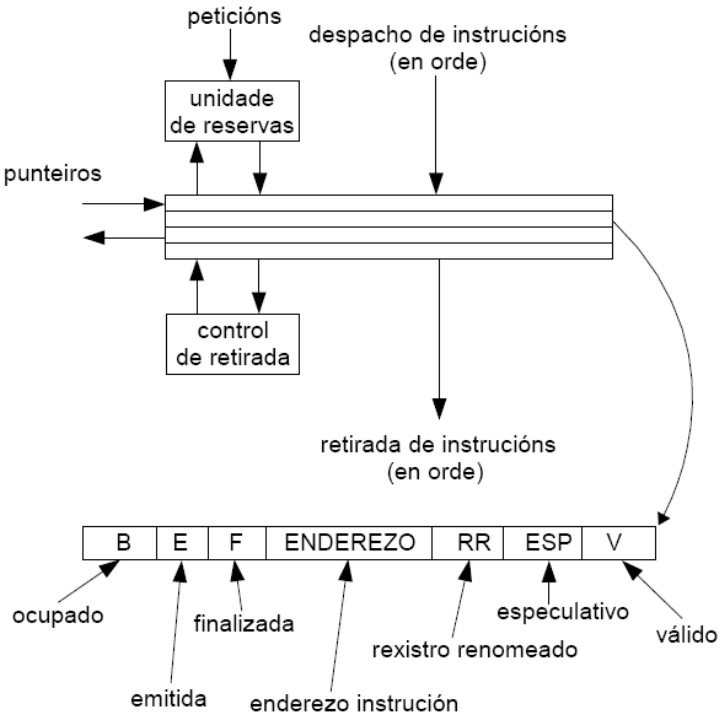


Figura 17: Arquitectura interna do ROB.

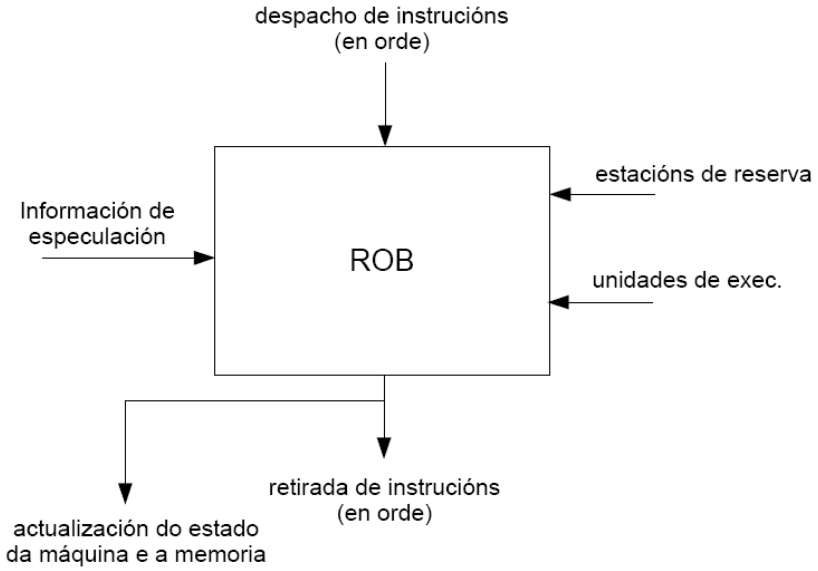


Figura 18: Interaccións co exterior do ROB.

4.4 Búfer de Reordenamento de Operacións de Memoria (MOB)

As operacións de lectura e escritura de memoria (Load e Stores) presentan algunhas diferencias no que atinxe ao tratamento de dependencias, con respecto a pares de instrucións que presentan dependencias a través de rexistros. A cuestión é que tanto para Loads como para Stores o cálculo do enderezo de memoria forma parte da súa execución, e polo tanto os enderezos de memoria non se coñecen ao decodificar a instrución. Deste xeito, non é trivial detectar dependencias e facilitar a execución de instrucións Load e Store fora de orde. O seguinte exemplo, ilustra esta situación:

ST R3 32(R1)

ADD R5 R3 R4

ST R5 67(R8)

....

LD R4 65(R6)

A instrución de Load almacenará o resultado no enderezo que resulte da suma do contido do rexistro R6 e o operando inmediato 65. Non se pode asegurar que este enderezo non será o mesmo que o enderezo de escritura en memoria dalgún das instrucións Store precedentes na orde do programa.

A execución fora de orde de instrucións de Load resulta fundamental para acadar maior paralelismo a nivel de instrución. A operación de Load é moito máis frecuente que a de Store, e usualmente marcan o inicio dun bloque de computación con dependencias asociadas aos valores que se cargan da memoria. Por outra banda, a posibilidade de ter fallos cache (e fallos de TLB, e fallos de páxina), implica que os Load poden ser operacións de elevada latencia en determinados casos. Polo tanto, resulta de interese anticipar o antes posible (en canto está dispoñible o rexistro base para o cálculo do enderezo) a execución de instrucións Load. Non obstante, os Loads lanzados fora de orde non alteran o estado da máquina até que son retiradas do ROB. Inicialmente len a memoria e cargan o valor nun rexistro renomeado. Só cando son retiradas, se produce a escritura no rexistro ISA. Deste xeito é posible executar instrucións Load de xeito especulativo (indicado no correspondente campo do ROB). A condición de retirada do ROB será que o estado xa no sexa especulativo e que sexa seguindo a orde do programa. Para cambiar un Load de estado especulativo a non especulativo, será necesario comparar o enderezo de lectura con todos os enderezos de escritura de instrucións Store que estaban presentes no ROB no momento en que entrou o Load (é dicir, chequear os Store que están a unha distancia como máximo correspondente ao número de entradas do ROB).

Os Stores modifican o estado da memoria e son irreversibles cando realizan a escritura. Polo tanto a escritura en memoria debe facerse seguindo a orde do programa. Para facilitar a execución fora de orde de instrucións de Store, engádese o búfer de reordenamento de operacións de memoria (MOB polas siglas en inglés). Por outra lado, o ancho de banda do sistema de memoria é escaso, e resulta fundamental darlle prioridade aos Load sobre os Store, xa que os Load son orixe de multitude de dependencias. Polo tanto as funcións do MOB son as seguintes:

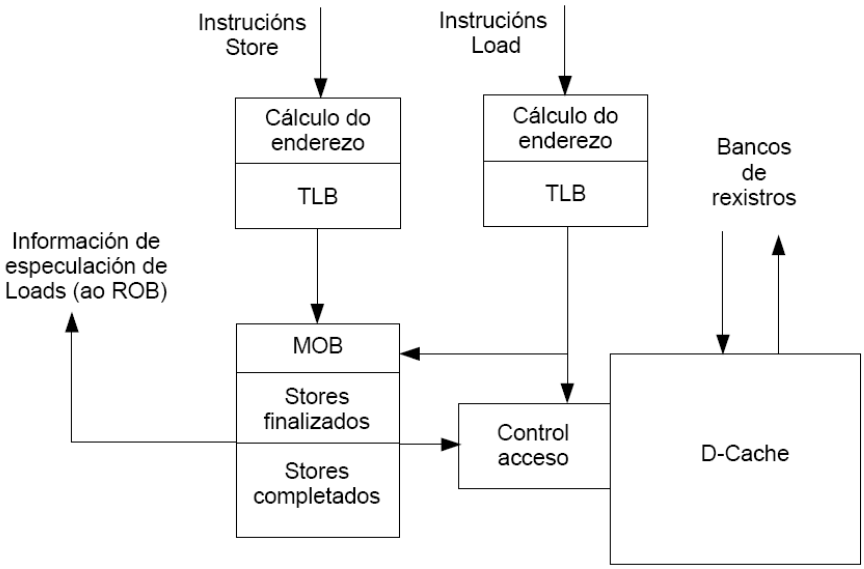


Figura 19: Ilustración das unidades de Load/Store con MOB.

- Almacenar nunha cola ordenada os resultados que se escribirán en memoria por instrucións Store xa completadas. Cando se dispoña de ancho de banda dispoñible coa memoria, realízanse as escrituras en orde dos Stores completados (retiradas dos Stores). Se ocorre unha excepción, antes do seu tratamento, estas escrituras de memoria pendentes teñen que levarse a cabo, xa que as instrucións correspondentes están completadas.
- Almacenar nunha cola ordenada, os resultados de instrucións Store finalizadas, pero aínda non completadas no ROB. Cando se retiran do ROB, pasan á cola das completadas.
- Chequear as instrucións de Load speculativas antes de ser retiradas e completadas: para isto é necesario comparar o enderezo de lectura do Load cos enderezos de escritura dos Store xa completados que están no MOB. O tamaño da cola de Stores completados debe ser suficiente para almacenar todos os Stores que potencialmente puideran estar presentes no ROB cando entrou a instrución de Load. Se o Load ten unha dependencia cun dos Stores da cola, entón debe descartarse o valor que se obtivo da memoria. Neste caso, unha opción é que o Load cargue directamente o valor producido polo Store que está na cola de completados, sen necesidade de acceder á memoria.

A Figura 19 ilustra dúas unidades para operacións de memoria, unha para Loads e outra para Stores, na que se incorpora o MOB. Logo do cálculo de enderezo, obtense o enderezo virtual, que debe ser traducido a un enderezo físico da memoria principal. Para realizar esta tradución accedese á TLB (Translation LookAside Búfer), que é unha cache do mapa de traducións. A TLB pode ter varios niveis, e se se produce un fallo en todos os niveis, é necesario acceder ao mapa de traducións na memoria principal (en caso de que non se atope no mapa de traducións, entón implica que a páxina non está na memoria principal, e que se produce un fallo de páxina). Dende o MOB debe saír información sobre o resultado de especulación dos Loads, que será utilizado polo ROB para poder completalos ou descartalos.

Polo tanto, o MOB desempeña un papel fundamental para acadar un maior palalelismo a nivel de instrución, facilitando a execución fora de orde de operacións Load e Store.

4.5 Arquitectura do Núcleo

Nesta subsección describimos a estrutura xeral do núcleo, composta polos elementos que describimos nas subseccións anteriores. A Figura 20 amosa a implementación de alto nivel cos diferentes bloques. Cada bloque pode corresponder a varias etapas (ciclos) do pipeline, de tal xeito que este pode ser bastante profundo.

Para unha implementación superscalar, en cada ciclo obtéñense un conxunto de instrucións da memoria cache. A lóxica de predición de saltos, detecta con elevada probabilidade os grupos nos que existen instrucións de salto que son tomados, e o seu enderezo de destino. Isto permite, especulativamente acceder a seguinte grupo de instrucións no seguinte ciclo. O deseño da memoria cache de instrucións non é trivial. Debe permitir a lectura dun elevado número de bytes por ciclo (varias instrucións). Por outra banda, o destino dunha instrución de salto, fai que os paquetes de instrucións non estean necesariamente aliñados dun xeito que coincida coa granularidade de acceso da memoria cache. Por outra banda, para repertorios de instrucións CISC (por exemplo as arquitecturas x86), existen diferentes tamaños para as instrucións, introducindo polo tanto complicacións adicionais.

Na etapa de decodificación, as instrucións son decodificadas en paralelo. Esta etapa pode resultar de elevada complexidade no caso de repertorios de instrucións CISC, debido a irregularidade do formato das instrucións. Unha estratexia común é descompoñer as instrucións CISC en microoperacións RISC elementais, que permiten unha decodificación e unha execución máis eficiente.

Logo da decodificación, realízase a etapa de renomeado de rexistros, lectura dos rexistros de entrada de cada instrución e o despacho ás estacións de reserva. Tamén se reserva unha entrada no ROB. Todo isto faise seguindo a orde das instrucións no programa. Para implementacións superscalares, o incremento do número de instrucións en paralelo complica todos estas accións dun xeito moi significativo.

Logo de despachar cada instrución a súa correspondente estación de reserva (neste exemplo eleximos unha implementación distribuída das estacións de reserva), as instrucións esperan a ter listos os operandos de entrada. Estes xa puideron ser lidos dos rexistros ISA ou dos non ISA se a referencia estaba renomeada e era válida. En caso contrario, a entrada que ocupa a instrución na estación de reserva almacena a referencia a un rexistro non ISA que está pendente de recibir un resultado dunha unidade de execución. As estacións de reserva emiten as instrucións que están listas ás unidades de execución. O ROB recibe información dos diferentes cambios de estado das instrucións, para almacenalo na entrada correspondente.

As unidades de execución poden ser de moitos tipos. Na Figura incorporarmos unha unidade para as instrucións de salto, outra para cálculos con enteiros, outra para cálculos de punto flotante, e dúas para operacións de memoria: unha para instrucións de Load e outra para instrucións de Store.

O ROB recibe información de cando cada instrución termina a execución, e no caso de ser especulativa, se a especulación foi correcta ou no. Se a especulación resultou ser correcta, a instrución xa poderá completarse, retirandoa do ROB (eliminando a entrada, e dando orden de que se actualice o rexistro ISA co valor do rexistro non ISA no que se renomeara, ou escribindo na parte de instrucións completadas do MOB, o valor que debe escribirse en memoria), seguindo a orde do programa.

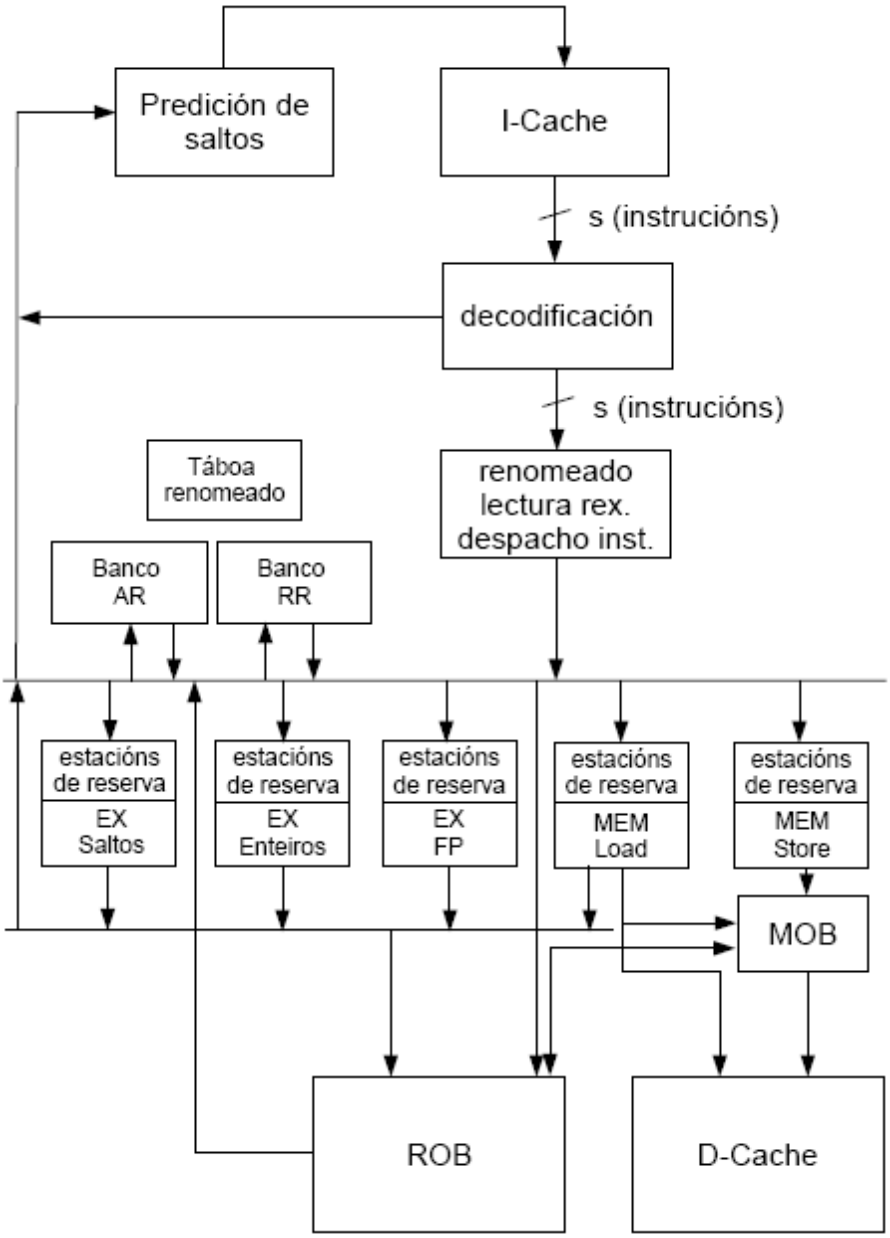


Figura 20: Arquitectura de alto nivel do núcleo.

4.6 Arquitectura do Núcleo Intel Nehalem

A Figura 21 amosa a arquitectura do núcleo correspondente á arquitectura de **Intel Nehalem**. O núcelo soporta dous threads hardware simultáneos, de tal xeito que se intenta manter ocupadas a unidades de execución con instrucións dos dous threads. Para soportar dous threads simultáneos, é necesario duplicar certas estruturas, coma por exemplo todos elementos que gardan o estado de cada thread.

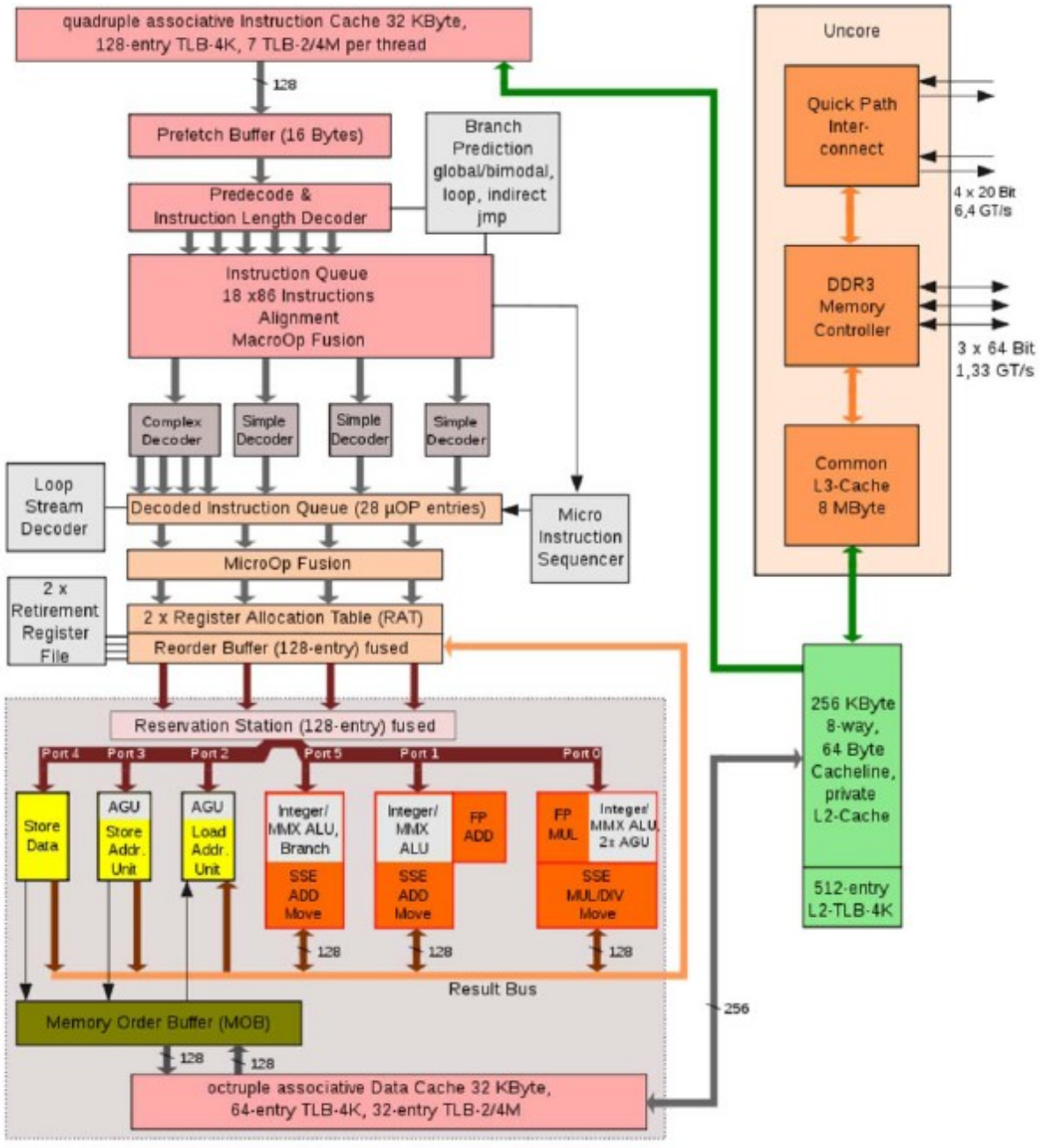


Figura 21: Arquitectura do núcleo no procesador Intel Nehalem.

Os diferentes subsistemas da arquitectura son os seguintes:

Sistema de Caches e TLBs:

Cada núcleo consta dous niveis de cache privada, e un terceiro nivel compartido por todos os núcleos. No nivel 1, cada núcleo ten unha cache de instrucións de outra de datos, asociativa de catro vías, de 32KB cada unha. No nivel 2, cada núcleo conta cunha cache unificada de datos e instrucións de 256 KB, asociativa de 8 vías. Finalmente, a cache de nivel 3, é compartida entre todos os núcleos (con 8 MB e asociatividade de 16 vías para este exemplo concreto).

As TLB permiten a tradución rápida dentro do chip dos enderezos virtuais a enderezos físicos (é en realidade unha cache da táboa de tradución de enderezos do sistema de memoria virtual). Esta arquitectura conta con dous niveis de TLB. No primeiro nivel consta de TLBs separadas para instrucións e datos, e tamén segundo o tamaño das páxinas. Para instrucións consta dunha TLB, compartida entre os dous threads, de 128 entradas (asociativa de 4 vías) para páxinas de 4KB, e unha TLB por thread, totalmente asociativa, de 7 entradas, para páxinas de 2/4MB. Para datos consta dunha TLB de 64 entradas para páxinas de 4KB, e outra de 32 entradas para páxinas de 2/4MB, en ambos casos compartidas para os dous threads, e asociatividade de 4 vías. A TLB de segundo nivel, é unificada para instrucións e datos, e consta de 512 entradas, con asociatividade de 4 vías, e para páxinas de 4KB.

Hardware de lectura e decodificación de instrucións

O núcleo conta cun sistema completo para predición de saltos (nas diferentes variantes que estudamos nas seccións anteriores: predición de salto tomado/non tomado, predición do enderezo de salto, predición de enderezo de retorno de chamadas, etc), que lle permite especular cal será o seguinte grupo de instrucións que se debe ler da cache. A cache ten unha interface de 16 bytes (128 bits) para lectura do grupo de instrucións. Estas instrución cárganse nun rexistro que ten esta capacidade (indicado na Figura 21 como “prefetch buffer”).

Como xa mencionamos anteriormente, a arquitectura do repertorio de instrucións x86 é de tipo CISC, co que as instrucións teñen formatos irregulares, e con instrucións que combinan acceso a memoria con computación. Para unha execución eficiente, estas instrucións complexas CISC son decodificadas en microoperacións, que emulan os repertorios de instrucións RISC, con formatos regulares, operacións aritméticas rexistro a rexistro, e operacións de memoria do tipo rexistro a memoria (ou memoria a rexistro). Polo tanto a etapa de decodificación nesta arquitectura é particularmente complexa.

Nunha primeira etapa da decodificación, estímase a lonxitude en bytes de cada instrución que foi lida da memoria. Tamén se comenza con parte do traballo de decodificación para transformalas en microoperacións. Ademais realízase o que Intel chama “Macro Op Fusion”, que consiste en xuntar dúas ou máis instrucións x86 cando a súa funcionalidade conxunta é equivalente a unha operación RISC. Esta etapa está indicada na Figura 21 coma “Predecode and Instruction Length Decoder”. A saída deste bloque ten seis portos, para seis instrucións x86. Estas van a unha cola de instrucións, con capacidade até 18 instrucións. Dende esta cola é posible decodificar até 4 instrucións x86, tres simples (digamos as que dan lugar a unha sola microoperación) e unha complexa (as que dan lugar até catro microoperacións). Se a instrución x86 é tan complexa que da lugar a un número maior de microoperacións, entón utilízase un microcontrolador (indicado coma “microinstruction sequencer” na Figura 21) para realizar a decodificación en varios ciclos (este proceso é lento, pero esta situación é pouco frecuente).

As microperacións obtidas pasan a unha cola, con capacidade para 28 entradas. Operando en paralelo a esta cola está o bloque para detección de lazos (“loop stream detector”), que analiza o

contido da cola de microoperacións para detectar se se trata dun lazo. En caso de detectar un lazo, non será necesario ler e decodificar repetidamente as instrucións do lazo, xa que están xa na cola de microoperacións.

Desta cola saen até catro microoperacións a un bloque para detectar a posible fusión de microoperacións ("MicroOp fusión") nunha soa operación que poida ser executada eficientemente polas unidades de execución.

Hardware para execución fora de orde e retirada en orde

Logo da fusión de microoperacións, poden lanzarse até catro delas á etapa de renomeado de rexistros (indicado coma RAT na Figura 21), e logo introdúcense no ROB e pasan as estación de reserva (RS). O ROB admite un total de 128 entradas para microoperacións, e inclúe os rexistros non ISA. A RS é unificada de 36 entradas (na Figura 21 hai un erro, xa que indica 128), con 4 portos para entrada de microoperacións, e cinco portos para emisión de microoperacións ás unidades de execución, e un porto para escrituras de datos en memoria (datos a escribir polos Stores). Dúas unidades de execución son para cálculo de enderezos, unha para Loads e outra para Stores. Logo existen tres conglomerados de unidades de execución, que fan diferentes tipos de operacións: enteiros, punto flotante, extensións multimedia (MMX e SSE), procesamento de instrucións de salto, etc. O MOB comunica coas tres unidades asociadas a operacións de memoria, e comunica tamén coa cache de datos, con dous buses de 128 bits, un para escritura e outro para lectura. O MOB conta con 48 búfers para almacenamento de Loads e 32 para almacenamento de Stores.

Todas as unidades funcionais están conectadas a un bus común, onde escriben os resultados, e fan posible actualizar os contidos das entradas do ROB e a RS. Como vemos existen dous bancos de rexistros ISA (un por thread), no que se permiten retirar até catro microoperacións por ciclo (na Figura 21 indícase coma "Retirement register File").

Aínda que non é un dato oficial, estímase que a profundidade deste pipeline é de 16 etapas, e como permite retirar até 4 microperacións por ciclo, o límite ideal de CPI é de 0.25 (con respecto as microoperacións, non con respecto as instrucións x86). Loxicamente, o CPI real será bastante superior.

A Figura 22 ilustra a seguinte xeración, o núcleo dos procesadores da familia **Intel "Sandy Bridge"**. Como elementos novidosos podemos destacar a inclusión dunha caché para microoperacións que permite evitar o proceso de decodificación das instrucións CISC se o resultado desta decodificación está na devandita caché. O ROB pasa de 128 entradas até 168. Un aspecto importante é que nesta implementación o ROB non incorpora os rexistros non ISA. Tampouco se ten un banco dedicado a rexistros ISA ("Retirement register file"). Neste caso temos dous bancos de rexistros (un para punto flotante e outro para punto fixo) que poden utilizarse tanto para rexistros ISA coma non ISA. Cada rexistro leva un bit de estado que indica se é un rexistro ISA ou non ISA. Para pasar dun a outro só é necesario cambiar dito bit de estado, se necesidade de mover os datos (do ROB ao banco de rexistros ISA). A estación de reserva é unificada, pero pasa de 36 a 54 entradas. O MOB conta con 64 búfers para almacenamento de Loads e 36 para almacenamento de Stores (48 e 32 para a arquitectura anterior). Finalmente nas unidades de execución, incorpora soporte para a execución das extensións AVX, que permiten operación vectoriais sobre 256 bits (128 bits na arquitectura anterior).

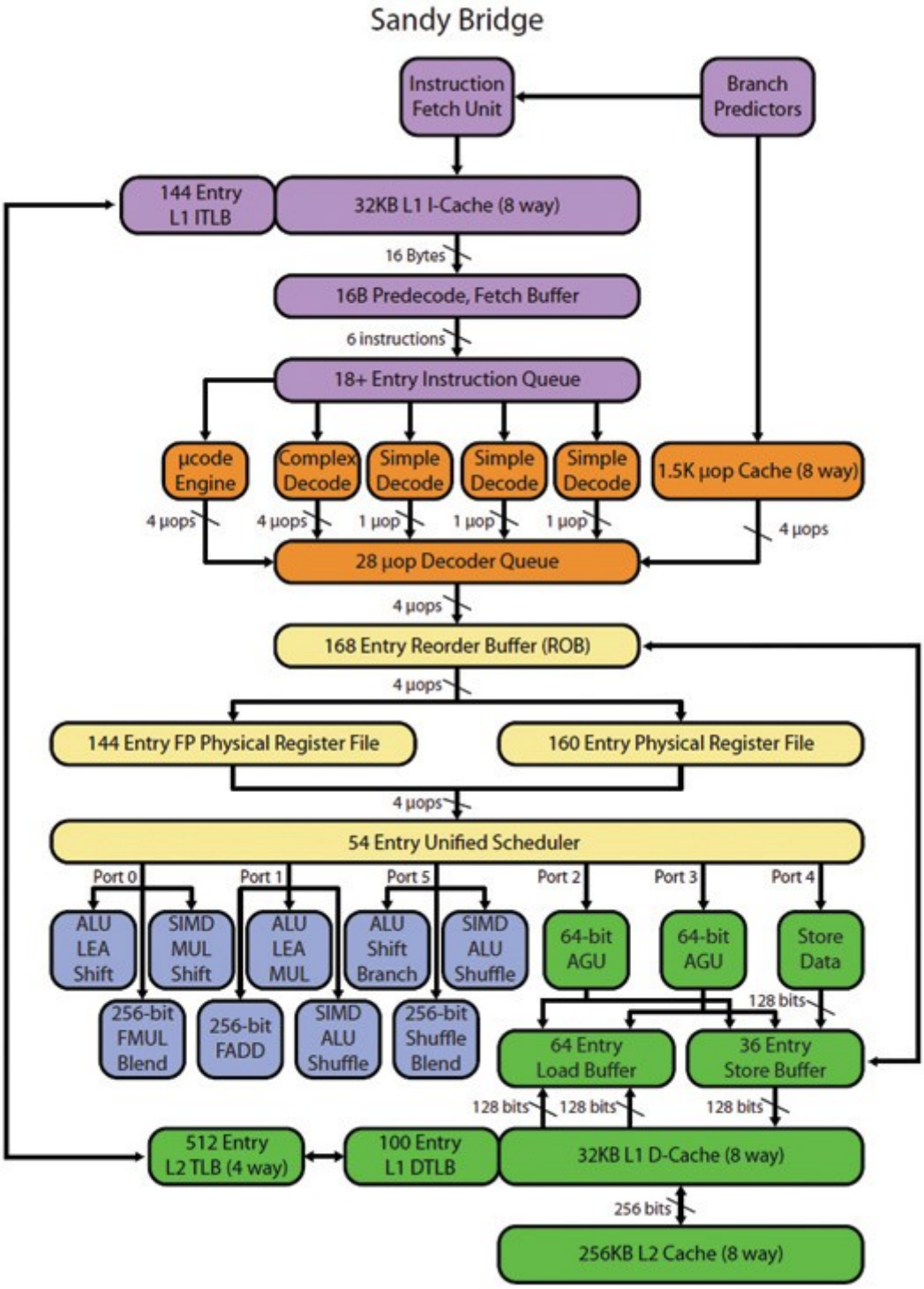


Figura 22: Arquitectura do Núcleo da Familia de Procesadores Intel Sandy Bridge

4.7 Escalamento dos núcleos

Os núcleos de procesamento presentan importantes barreiras para o seu escalamento. En xeral pódese establecer que as prestacións dun núcleo escalan coa raíz cadrada do seu tamaño. Isto é debido a que para diminuir o CPI, é necesario utilizar unha mellor predición de saltos para poder lanzas máis instrucións en paralelo, o que implica un maior número de unidades de execución, maior tamaño das estacións de reserva, do ROB, etc. As estruturas coma as estacións de reserva ou o ROB, escalan mal co número de entradas, e tamén co número de portos de lectura/escritura. Isto é debido a que son estruturas de memoria totalmente asociativas, na que as buscas deben facerse comparando con todas as entradas presentes.

Polo tanto, aínda que podemos esperar algunha melloras nos núcleos, é de esperar que non reduzan significativamente o CPI no futuro, e que o escalamento fundamental veña polo número de núcleos.

4.8 Arquitecturas superscalares alternativas: VLIW

As arquitecturas VLIW (por Very Long Instruction Word en inglés) baséanse en que o compilador intente obter gran parte do paralelismo a nivel de instrución. O compilador agrupa varias instrucións nunha instrución grande, para logo en tempo de execución, lanzalas simultaneamente á execución. Polo tanto as instrucións que poden ir nunha instrución grande están restrinxidas polos diferentes tipos de conflitos que poden darse (estruturais, dependencias, etc).

O compilador debe utilizar técnicas coma desenrolo de lazos (poñer varias iteracións dun lazo coma unha secuencia de instrucións) e outras máis avanzadas, para poder atopar as instrucións que pode lanzar en paralelo. En arquitecturas VLIW avanzadas, como son os Intel Itanium, incorporan mecanismos hardware que complementan a labor do compilador, e tamén técnicas de especulación, sobre todo para saltos. Problemas tradicionais deste tipo de procesadores son: códigos moi grandes (xa que nalgunha palabras non será posible incluír todas as instrucións que pode albergar), necesidade de recompilación se cambian o número de unidades funcionais ou a súa disposición, e a falta de información en tempo de compilación para detectar o paralelismo a nivel de instrución. As tecnoloxías desenvolvidas para os Intel Itanium intentaron paliar en certa medida estas dificultades.

Este tipo de arquitecturas ten bastante éxito nos procesadores empotrados adicados a tarefas multimedia, no que o paralelismo a nivel de instrución en abundante e fácil de atopar por parte do compilador. Non obstante, para este tipo de cargas de traballo, atopan un forte competidor nos procesadores vectoriais é de streaming.