Arquitectura de Computadores

Notas de Clase

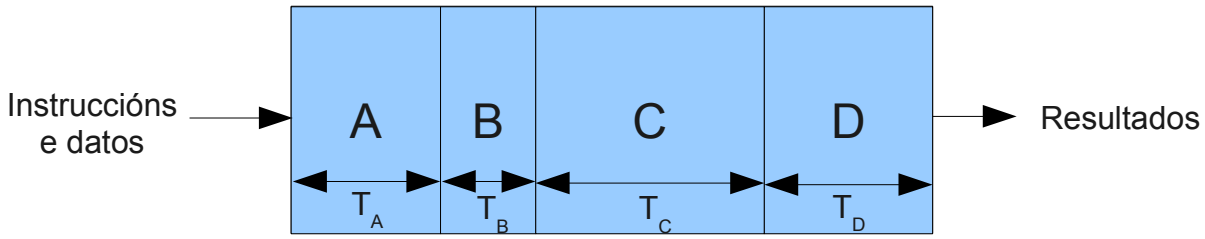
Elisardo Antelo Suárez

Tema 3: Núcleos de Procesamento - Segmentación (Pipelining)

1- Introdución

Neste tema e no seguinte estudaremos en profundidade a estrutura e arquitectura dos núcleos de procesamento. Logo de ter introducido a estrutura básica dun elemento de procesamento (control e camiño de datos) en materias anteriores, nestes dous temas introduciremos técnicas que permiten explotar o paralelismo a nivel de instrución (procesamento paralelo de instrucións). Neste tema introducimos os conceptos básicos para entender unha das técnicas máis potentes para explotar o paralelismo a nivel de instrución: a segmentación (en inglés "pipelining"). No tema seguinte introduciremos conceptos máis avanzados, que permitirán entender a arquitectura dos núcleos de procesamento que se utilizan actualmente nos microprocesadores comerciais.

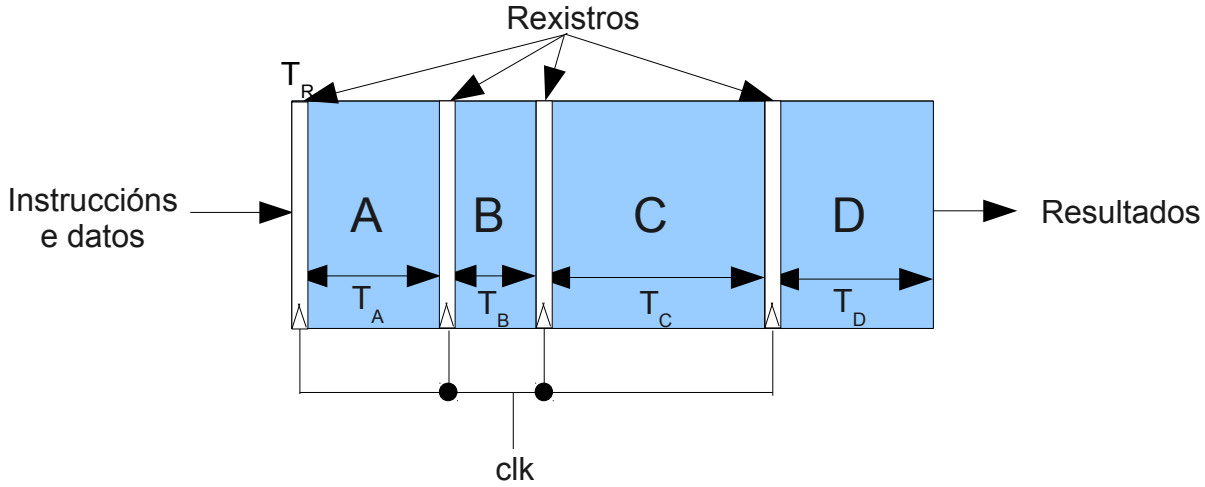
Un núcleo de procesamento recibe instrucións e datos (almacenados nas memorias cache), e permite obter os resultados intermedios que serán utilizados por outras instrucións xunto con outros datos. Para isto é necesario un circuíto combinacional (camiño de datos e control) bastante complexo que conta con varios subsistemas ben definidos. Por exemplo, na Figura 1 amósase un núcleo cos subsistemas A, B, C e D. Estes subsistemas realizan algunha das funcións necesarias para executar as instrucións (cos datos) que entran no bloque combinacional.



$$T_{COMB}=T_A + T_B + T_C + T_D$$

Figura 1: Abstración dun núcleo de procesamento coma un bloque combinacional.

Esta estrutura permite obter un resultado en cada instante de tempo T_{COMB} (retardo máximo do circuíto combinacional). Unha posible mellora desta estrutura consiste en introducir rexistros entre os diferentes bloques, para aillalos electricamente. Deste xeito, nun instante dado, teremos unha instrución distinta en execución en cada un dos bloques, e podemos retirar instrucións a un ritmo superior que no sistema da Figura 1. A Figura 2 amosa o sistema logo da introdución de rexistros entre bloques. Os rexistros están sincronizados por un sinal de reloxo (clk) de tal xeito que no flanco positivo deste sinal, os rexistros cárganse cos sinais producidos no bloque anterior.



$$T_{Clk}=\max\{T_R+T_A, T_R+T_B, T_R+T_C, T_R+T_D\}$$

Figura 2: Núcleo segmentado por bloques con rexistros

O ciclo de reloxio (retardo entre flancos positivos do sinal clk) deber ser suficiente para permitir a cada bloque calcular o seu resultado (para o peor caso, é dicir, o camiño de sinal máis longo) e cargalo no rexistro. Polo tanto o ciclo de reloxo será o máximo dos retardos dos bloques A, B, C ou D, máis o retardo de cargar o rexistro. Isto permite, en xeral, obter resultados a un ritmo moi superior ao do núcleo ilustrado na Figura 1 (sen rexistros intermedios). Por exemplo, imaxinemos que o retardo dos bloques é o seguinte: bloque A (25 FO4), bloque B (15 FO4), bloque C (35 FO4) e bloque D (27 FO4). Para o rexistro asumimos un retardo de 3 FO4. Polo tanto o núcleo da Figura 1 permite obter, no mellor caso, un resultado por cada 25+15+35+27=101 FO4. O núcleo da Figura 2, permite obter, no mellor caso, un resultado por cada max{25,15,35,27} + 3=38 FO4.

Polo tanto, a segmentación permite solapar a execución de diferentes instrucións e acelerar así o ritmo de produción de resultados ("throughput" en inglés). Nun instante dado, os bloques A, B, C e D, no exemplo anterior, están a resolver para distintas instrucións unha parte da computación que require cada unha delas. Isto pode expresarse en forma de diagrama de tempos, tal e como se amosa na Figura 3. Por exemplo, no ciclo 5, o bloque A está resolvendo parte da instrución I5, o bloque B parte da instrución I4, o bloque C parte da instrución I3, e o bloque D parte da instrución I2. Concluímos polo tanto que nese ciclo estase a solapar a execucións das instrucións I2, I3, I4 e I5. Como vemos existe unha latencia inicial até que comezamos a obter resultados (o tempo de encher as etapas do núcleo). Logo deses ciclos iniciais, é posible obter un resultado dunha instrución por ciclo (no mellor caso, xa que como veremos moi frecuentemente non é posible introducir unha instrución por ciclo na unidade).

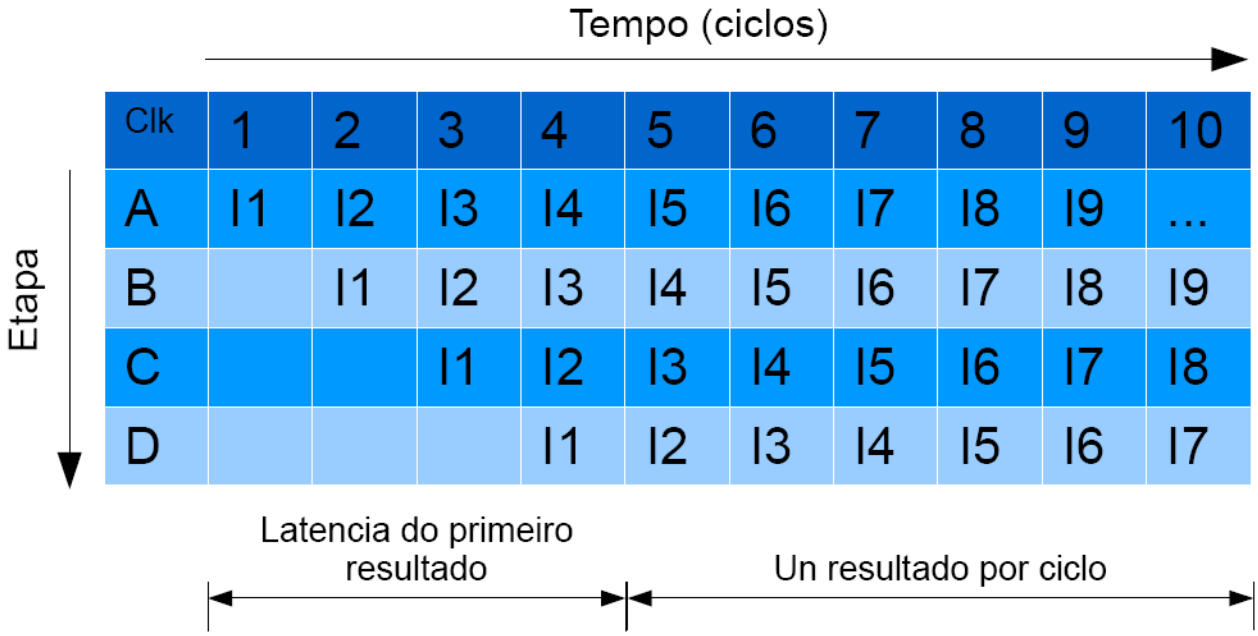


Figura 3: Ilustración da execución en pipeline.

2- Gañancia en velocidade coa segmentación e limitacións

Para reducir o tempo de ciclo é posible balancear as etapas do pipeline. Por exemplo, no caso da Figura 2, é posible modificar o emprazamento dos rexistros para intentar que cada etapa teña un retardo semellante (o caso ideal sería que todas as etapas tivesen o mesmo retardo, pero isto é moi complicado de acadar nas implementacións prácticas), co que se consegue unha redución do tempo

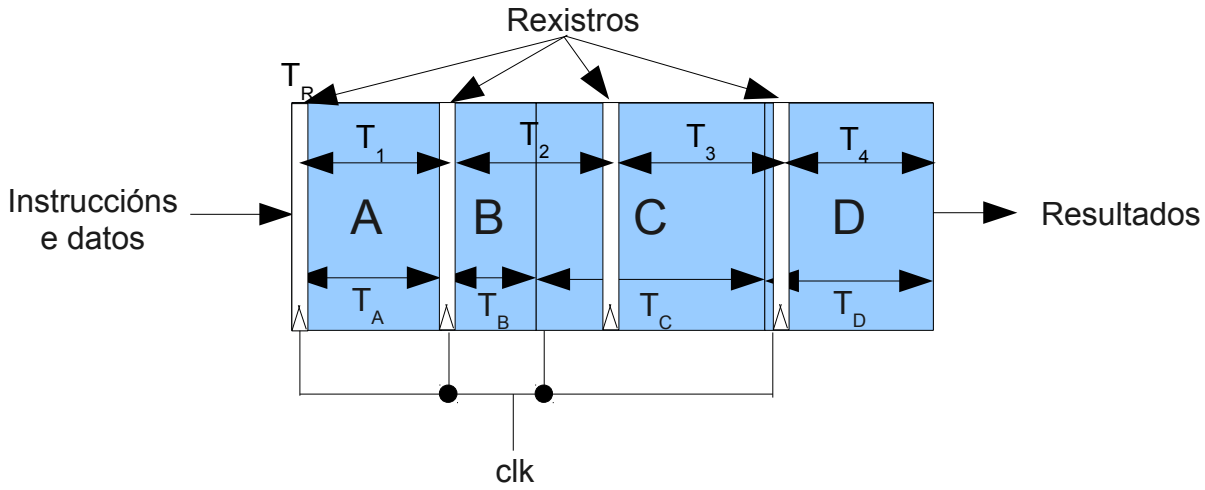


Figura 4: Emprazamento de rexistros para equilibrar as etapas do pipeline.

de ciclo, xa que a latencia da etapa máis lenta redúcese. A Figura 4 ilustra o novo emprazamento dos rexistros.

No caso ideal os retardos de cada etapa T1, T2, T3 e T4 serían iguais a T_R+T_S, con $T_S=(T_A+T_B+T_C+T_D)/4$. Non sempre será posible acadar un balanceo ideal no retardo de cada etapa, polo que en realidade T_S será un pouco superior. No exemplo que estamos a desenvolver, a cota inferior para T_S é 25.5 FO4, polo que a cota mínima para o tempo de ciclo neste pipeline é de 28.5 FO4, incluíndo o retardo asociado ao rexistro.

Nos programas reais non sempre será posible atopar unha instrución para introducila no pipeline en cada ciclo. As razóns poden ser moi variadas:

i) Os datos non están dispoñibles: isto pode ser debido a un fallo de cache, ou o dato ten que ser actualizado por instrucións precedentes (dependencias de datos) que aínda non se retiraron do pipeline.
ii) As instrucións de salto condicional: estas deben rematar a súa execución no pipeline para coñecer cal é a seguinte instrución a executar. Por un lado debe determinarse se se cumpre a condición de salto (se non se cumpre a seguinte instrución será a que segue a orden secuencial), e tamén cal é o enderezo da instrución de destino do salto (se a condición é verdadeira).

Para caracterizar este efecto, introducimos un parametro P, que indica a probabilidade por ciclo do pipeline de retirar o resultado dunha instrución. Por exemplo P=0.9, indica que no 90% dos ciclos obtemos un resultado dunha instrución como saída do pipeline, e que no 10% dos ciclos non se computa nada na saída (xa que non se emitiron as correspondentes instrucións neses ciclos). Este tipo de ciclos nos que non se computa nada denomínanse burbullas do pipeline ("bubbles" en inglés), e corresponden ao caso no que no pipeline se introduce un código de instrución correspondente a unha non operación (NOP).

Un aspecto importante é coñecer a ganancia en velocidade que resulta logo de incluír rexistros para configurar un pipeline. Para una total de I instrucións executadas a ganancia en velocidade da versión pipeline do núcleo de procesamento está dada por:

$$S = \frac{I \times T_{COMB}}{\left((N-1) + \frac{I}{P}\right) \times (T_R + T_S + T_{OV})}$$

onde:

I x T_{COMB}: tempo de computación da versión sen rexistros de pipeline.

N-1: número de ciclos necesarios para encher un pipeline de N etapas (no exemplo da Figura 3 este valor corresponde a 3 ciclos).

I/P: número de ciclos necesarios para obter o resultado de I instrucións logo de encher o pipeline. Por exemplo para P=0.91, estatisticamente serán necesarios aproximadamente 1.10 x I ciclos para obter o resultado das I instrucións.

T_R: retardo do rexistro (inclúe a sobrecarga por sincronización, etc).

T_S: é o retardo por etapa ideal correspondente á parte combinacional. Polo tanto para un pipeline de N etapas, este valor é $T_S = T_{COMB}/N$.

T_{OV}: é o tempo adicional do ciclo de reloxio debido a un balanceo non ideal das etapas do pipeline.

A ganancia en velocidade pode expresarse dun xeito máis conveniente como

$$S = \frac{N}{\frac{1}{P} + \frac{(N-1)}{I} + \left(\frac{1}{P} + \frac{(N-1)}{I}\right) \times \left(\frac{T_R + T_{OV}}{T_{COMB}/N}\right)}$$

Normalmente ten sentido utilizar unha estrutura en pipeline cando I>>N, polo tanto neste caso de interese práctico, a ganancia en velocidade pode aproximarse por ((N-1)/I -> 0)

$$S = \left(\frac{P}{1 + \frac{(T_R + T_{OV})}{T_{COMB}/N}}\right) \times N$$

P depende de N, xa que canto máis profundo é o pipeline (N maior), a incidencia das dependencias e dos saltos será maior (tárdase un maior número de ciclos en resolver as dependencias ou os saltos), e o valor de P diminúe. A Figura 5 ilustra de xeito cualitativo a dependencia de P con N.

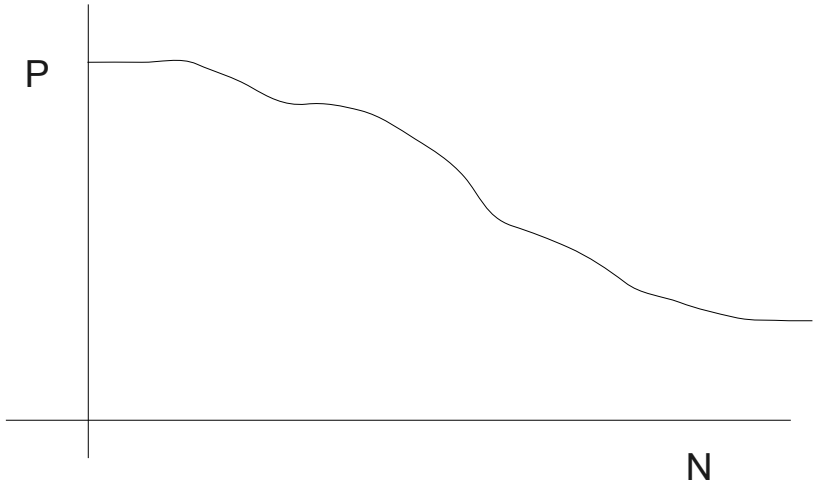


Figura 5: Dependencia cualitativa entre P e N

Por outro lado, o termo $(T_R + T_{OV})/(T_{COMB}/N)$ aumenta con N, e representa a sobrecarga por introducir rexistros, é dicir, a sobrecarga asociada á configuración en pipeline fronte ao traballo útil que se fai en cada etapa. Se N medra, o traballo útil por etapa decrece e tende a reducir a vantaxe de utilizar un pipeline (o retardo do rexistro, de sincronización e a sobrecarga de desbalanceo das etapas do pipeline faise importante fronte ao retardo do bloque combinacional). Notar que o aumento de frecuencia non é estritamente lineal co número de etapas, debido ao termo constante de sobrecarga no tempo de ciclo. Cando o termo de sobrecarga é comparable ao retardo combinacional de cada etapa, entón o aumento da frecuencia co número de etapas é moi limitado. Polo tanto, existe un máximo da ganacia en velocidade para un determinado valor N. A Figura 6 ilustra a forma da función S(N). Para valores baixos de N, S aumenta ao aumentar o número de etapas, xa que o que domina é o solapamento entre instrucións. Para valores de N altos, S comenza a reducirse xa que o efecto positivo do solapamento de instrucións non é suficiente para contrarrestar o efecto negativo en P por un pipeline moi profundo, e polo peso cada vez maior do retardo non útil (rexistro, sincronización e balanceo de etapas). Ademais debe terse en conta o custe (en área e potencia) dos propios rexistros para configurar o pipeline, e a súa sincronización (distribución do sinal de reloxio a cada rexistro). Para pipelines moi profundos (valor elevado de N) o número de rexistros será elevado e a frecuencia do sinal de reloxo tamén será elevada (xa que o retardo de etapa decrece con N), polo que ten un dobre efecto no consumo de potencia debido á configuración en pipeline.

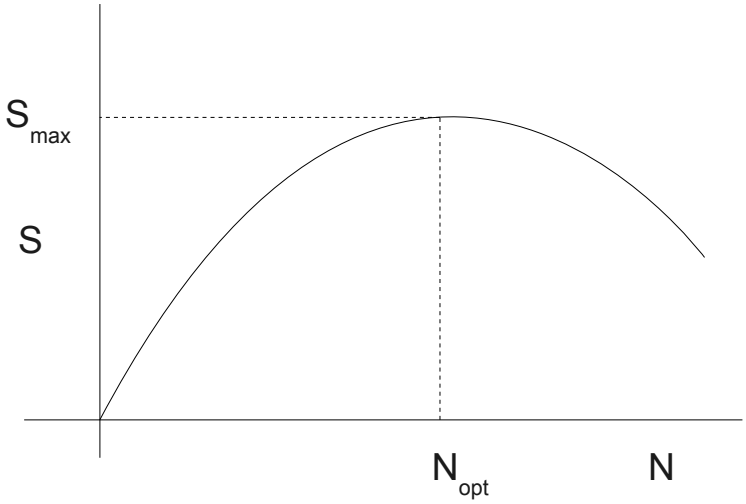


Figura 6: Dependencia cualitativa de S con N.

No contexto actual, os pipelines dos núcleos optimízanse normalmente para maximizar a relación (S^3/Potencia) (onde Potencia é a suma da potencia estática e dinámica no núcleo), dado un límite para o consumo máximo de potencia.

Resulta interesante descompoñer os factores que contribúen a P, e identificalos cos diferentes eventos que fan que se insiran burbullas no pipeline. Estes eventos de conflito denomínanse en inglés "hazards". Podemos atopar os seguintes tipo de conflitos:

- Conflitos estruturais: cando dúas instrucións necesitan a mesma unidade hardware para completar unha etapa do pipeline, só unha delas pode avanzar. A outra instrución queda parada á espera de poder seguir coa execución. Se as instrucións son emitidas en orde, e unha a unha, isto insire burbullas no pipeline. Chamarémoslle P_S a probabilidade por instrución de ocorrencia dun factor estrutural, e C_S o custo en ciclos asociado a este conflito, é dicir o número de ciclos que se insiren burbullas (NOP) por cada conflito estrutural.
- Conflitos por dependencia de datos (xa comentados anteriormente). P_D e C_D son respectivamente as probabilidades por instrución dun conflito por dependencia de datos e o custo en ciclos asociado a dito conflito.
- Conflitos por control de fluxo. Estes son os debidos a saltos condicionais, como xa comentamos anteriormente (tamén poden corresponder a saltos incondicionais, pero que necesitan calcular o enderezo de destino do salto). P_B e C_D son os correspondentes parámetros para este tipo de conflito.

Polo tanto a probabilidade de retirar unha instrución por ciclo está dada por

$$P=\frac{I}{I+P_S\times I\times C_S+P_D\times I\times C_D+P_B\times I\times C_B}=\frac{1}{1+P_S\times C_S+P_D\times C_D+P_B\times C_B}$$

O denominador indica o número de ciclos (logo de encher o pipeline) necesarios para completar a execución de I instrucións. En realidade P é coñecido como o IPC: instrucións retiradas por ciclo do pipeline, e o seu inverso é o CPI (=1/P), o número medio de ciclos por instrución. Polo tanto podemos expresar a ganancia en velocidade en termos do CPI, do seguinte xeito:

$$S=\frac{N}{CPI\times\left(1+\frac{T_R+T_{OV}}{T_{COMB}/N}\right)}$$

$$CPI=1+P_S\times C_S+P_D\times C_D+P_B\times C_D$$

Por exemplo, se a probabilidade dun conflicto estrutural é de 0.05 cun custo de 10 ciclos, a probabilidade dun conflicto por dependencia de datos é de 0.25 cun custo de 7 ciclos, e a probabilidade dun conflito de control é de 0.20 cun custo de 4 ciclos, entón o CPI resultante é de:

$$CPI=1+0.05\times 10+0.25\times 7+0.20\times 4=4.05$$

É dicir, para un pipeline que ten a capacidade de emitir unha instrución por ciclo, tendo en conta as penalizacións anteriores polos diferentes tipos de conflitos, resulta que acada unha produtividade aproximada de unha instrución cada catro ciclos. Notar que non se ten en conta as penalizacións por fallos nas memorias cache, é dicir que supoñemos que os datos e as instrucións están sempre dispoñibles para executalas no pipeline (é dicir, é o CPI do núcleo ideal, como vimos no Tema 1).

3- Exemplo de pipeline (básico)

Nesta sección vamos a presentar a implementación dun pipeline básico para coñecer aspectos de baixo nivel. Para simplificar a presentación e o estudo, consideramos a implementación dun reducido número de instrucións:

i) Instrucións ALU: estas instrucións toman o contido de dous rexistros, ou dun rexistro e dun valor inmediato (codificado na propia instrución e estendido en signo ao ancho da ALU – por exemplo un operando inmediato de 16 bits e estendido ao ancho dunha ALU de 32 bits), realiza a operación na ALU que está codificada na instrución, e coloca o resultado no rexistro destino. Exemplos deste tipo de instrucións son (DADD, DSUB, DADDI, DSUBI, AND, OR, DADDU, DSUBU, DADDIU, DUSBIU, onde I indica que a instrución utiliza un operando inmediato, e U corresponde a operacións sen signo).

ii) Instrucións Load/Store: estas instrucións utilizan un rexistro e un valor inmediato para calcular o enderezo de memoria. Para Load indícase o rexistro no que se cargará o valor que se obtén da memoria, e para Store o rexistro onde está o valor que se almacenará en memoria.

iii) Bifurcacións e saltos no control do programa: A condición de salto pode especificarse con códigos de condición (que son o resultado de executar algunha instrución), ou mediante comparacións entre valores de dous rexistros, ou entre o valor dun rexistro e cero. Neste exemplo consideraremos comparación con cero do contido dun rexistro como condición de salto. O destino do salto está dado por un operando inmediato, que se lle suma ao contador do programa.

Na Figura 7 amósase un pipeline que implementa este repertorio de instrucións reducido. As etapas separadas por rexistros son: IF (busca da instrución), ID (decodificación da instrución), EX (execución), MEM (escritura/lectura da memoria), WB (postescritura). PC é o rexistro que almacena o enderezo de memoria da seguinte instrución. No resto desta sección analizaremos como se implementa cada tipo de instrución neste pipeline simple.

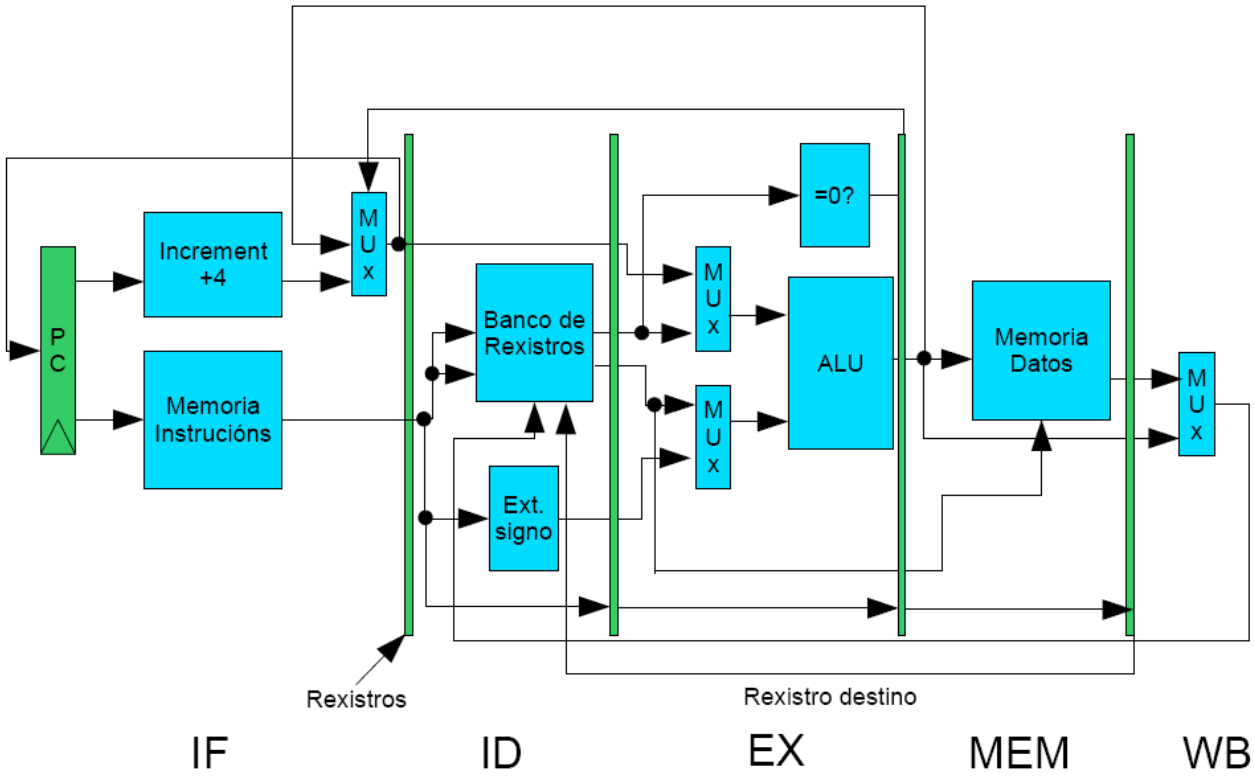


Figura 7: Arquitectura do Pipeline.

3.1 Implementación de instrucións tipo ALU

Estas instrucións son de rexistro a rexistro (é dicir os operandos orixe e destino están no banco de rexistros). No rexistro PC (contador do programa) temos almacenado o enderezo dunha instrución ALU. As accións nas diferentes etapas son do seguinte modo:

Etapa IF: Obtense a instrución da memoria (cache de instrucións) e pásase a entrada do rexistro da etapa ID. Increméntase o contido de PC por 4 (supoñendo instrucións de 32 bits), para obter o enderezo da seguinte instrución secuencial. No multiplexo seleccionase este enderezo ou outro corresponde a unha instrución de salto anterior (o sinal de control do multiplexo e a outra entrada proceden do rexistro da etapa MEM), e pásase a entrada do rexistro PC (seguinte instrución).

Etapa ID: decodifícase a instrución (para obter sinais de control, etc). En paralelo obtéñense os identificadores dos rexistros directamente da propia instrución, para a lectura do banco de rexistros. Supoñemos un banco de rexistros con dous portos de lectura e un de escritura Se un dos operandos é inmediato, realízase a extensión de signo para que teña o ancho completo da ALU. Ao fnal desta etapa, estarán dispoñibles os operandos sobre os que se quere operar.

Etapa EX: Os multiplexos seleccionan a entrada correspondente dos rexistros do pipeline, e realízase a operación na ALU. O valor do resultado cárgase no seguinte rexistro do pipeline (rexistro da etapa MEM).

Etapa MEM: non se realiza ningunha acción nesta etapa, para este tipo de instrucións.

Etapa WB: o multiplexo selecciona o resultado da ALU, que pasou sen modificar pola etapa MEM. O control selecciona o rexistro para escritura no banco de rexistros (a información de control da instrución vai pasando polos diferentes rexistros do pipeline). Observa que a etapa ID e WB comparten o banco de rexistros, pero utilizan tres portos independentes (dous de lectura na etapa ID e un de escritura na etapa WB).

3.2 Implementación de instrucións tipo Load/Store

Estas instrucións son de memoria a rexistro (Load) ou de rexistro a memoria (Store). As accións nas diferentes etapas son do seguinte modo:

Etapa IF: esencialmente igual ao caso anterior.

Etapa ID: decodifícase a instrución. Para os Stores, lectura de dous rexistros, que conteñen a base do enderezo e o valor que se quere almacenar; o operando inmediato é o desprazamento para o cálculo do enderezo de almacenamento, que é estendido en signo para operar na ALU. Para Loads, fai a lectura dun rexistro (base do enderezo) e esténdese en signo o operando inmediato, que é o desprazamento para o cálculo do enderezo.

Etapa EX: O multiplexo superior selecciona o contido do rexistro coa base do enderezo. O inferior selecciona o operando inmediato co signo estendido. A ALU realiza o cálculo do enderezo, sumando a base e o desprazamento. Para os Stores, o contido do rexistro a almacenar en memoria pasa por esta etapa sen modificar.

Etapa MEM: Realízase a escritura (Store) ou lectura (Load) da memoria (cache de datos), utilizando o enderezo calculado na ALU (e o contido do rexistro a almacenar en memoria procedente da etapa ID e que non se modificou na etapa EX). Este é o último ciclo para o Store.

Etapa WB: o multiplexo seleciona a saída da memoria e escribe o valor no banco de rexistros completando a instrución de Load.

3.3 Implementación de instrucións de salto

Esta instrución realiza a comparación de igualdade con cero. Inclúe un operando inmediato que constitúe o desprazamento do contador do programa para obter o enderezo de salto. As accións nas diferentes etapas son do seguinte modo:

Etapa IF: Semellante aos casos anteriores, salvo que no próximo ciclo, en principio, non se coñece o enderezo da seguinte instrución. Unha opción é facer unha parada na emisión de novas instrucións (en inglés chámase "stall"), até que se resolva a instrución de salto, e se coñeza o enderezo da seguinte instrución. Logo discutiremos diferentes opcións. A saída do mux logo de incrementar o PC por catro pasa a seguinte etapa, xa que será necesaria para calcular o enderezo do salto (se este se avalía como verdadeiro).

Etapa ID: Lectura do banco de rexistros para comparar o operando con cero, e extensión de signo do campo de desprazamento do enderezo para cálculo do enderezo do destino do salto. O enderezo do PC actual incrementado en 4 pasa a seguinte etapa.

Etapa Ex: comparación con cero do contido do rexistro para detectar se a condición de salto se verifica (salto tomado – "branch taken" en inglés), e súmase o campo de desprazamento ao enderezo PC+4, e calcúlase o enderezo do destino do salto á saída da ALU.

Etapa MEM: faise un traspaso do resultado da decisión de salto e do enderezo de salto á etapa IF, e selecciónase o enderezo para cargar no rexistro PC.

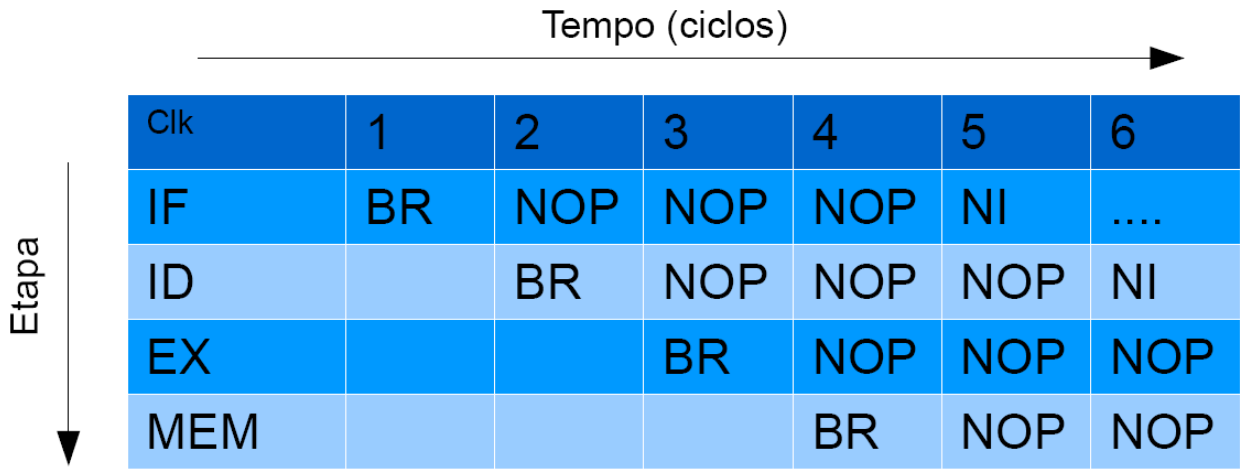


Figura 8: Coste dunha instrución de salto.

Na Figura 8 amósase o custo dunha instrución de salto (3 ciclos neste pipeline). Durante tres ciclos, emítense códigos de instrución NOP (Non Operación) na etapa IF até que no ciclo 4 xa se coñece o enderezo da seguinte instrución, que se obten no ciclo 5 (NI: seguinte instrución).

3.4 Melloras do Pipeline

Nesta subsección introduciremos un par de melloras no pipeline que permiten reducir as penalizacións por saltos e por certas dependencias. Especificamente pódese anticipar a resolución da instrución de salto e a determinación do enderezo do salto poñendo o hardware necesario na

etapa ID: o detector de cero e un sumador. Por outra banda é posible evitar paradas do pipeline debido a dependencias, incluíndo envío de datos entre etapas ("bypassing" ou "forwarding" en inglés). Por exemplo, é posible enviar un dato computado na ALU a unha das entradas da propia ALU no seguinte ciclo incluíndo unha conexión dende a etapa MEM (dende o propio rexistro) até os multiplexos da etapa EX. Deste xeito dúas instrucións ALU consecutivas con dependencia de datos non producirían parada no pipeline, xa que o resultado que necesita a segunda das instrucións está dispoñible xusto cando entra na etapa EX.

Na Figura 9 amósase a implementación do pipeline coas melloras indicadas. Incluímos envíos de datos dende as etapas MEM e WB á EX (entrada dos multiplexos).

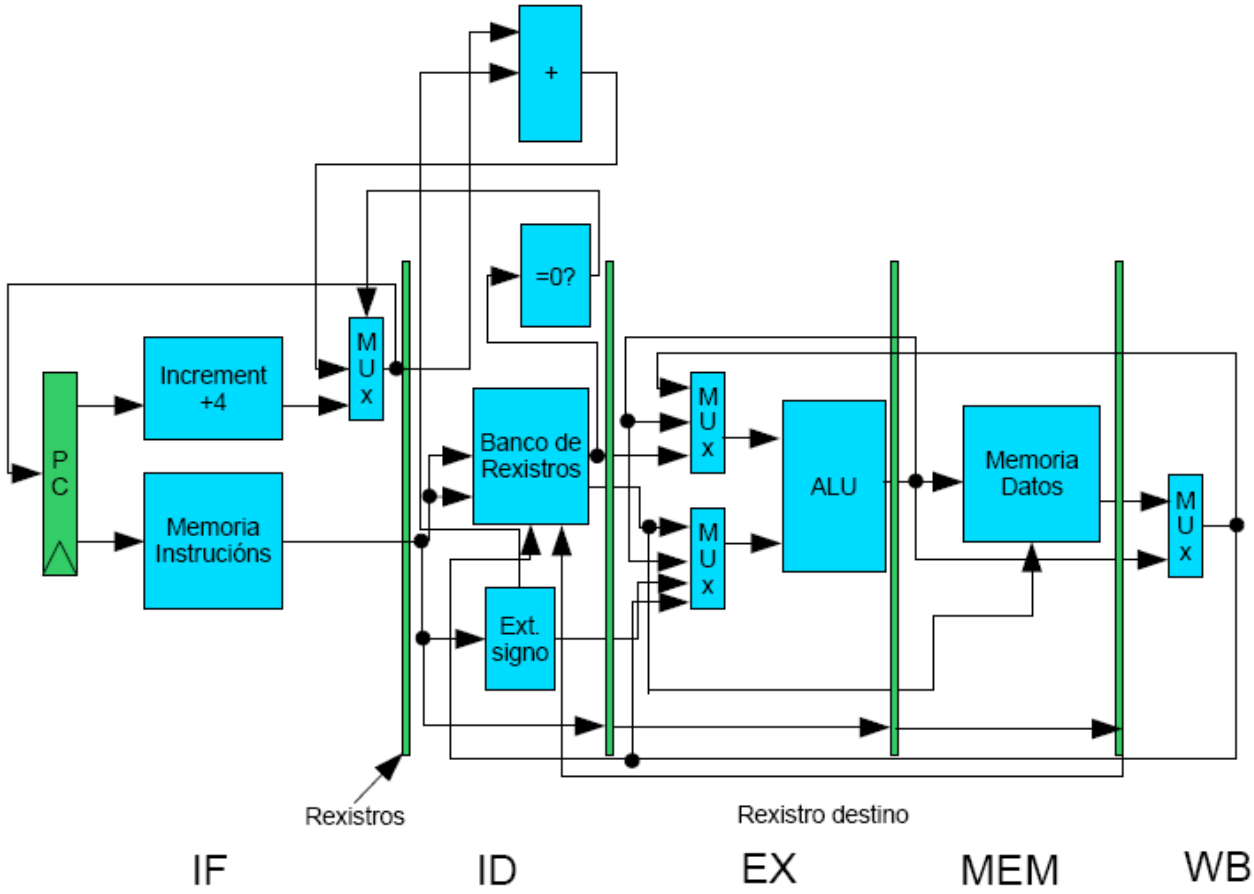


Figura 9: Pipeline coas modificacións para reducir penalizacións.

A introdución destas melloras supón un coste adicional en hardware, xa que se incorpora un sumador, increméntase o número de conexións polo envío de datos entre etapas, e aumenta o número de entradas dos multiplexos da etapa EX.

Na Figura 10 ilustramos a mellora na penalización por saltos no esquema da Figura 9. Coa mellora, a penalización da instrución de salto é de un ciclo (párase a emisión de instrucións logo da de salto durante un ciclo), en contraposición co esquema orixinal, cunha penalización de tres ciclos.

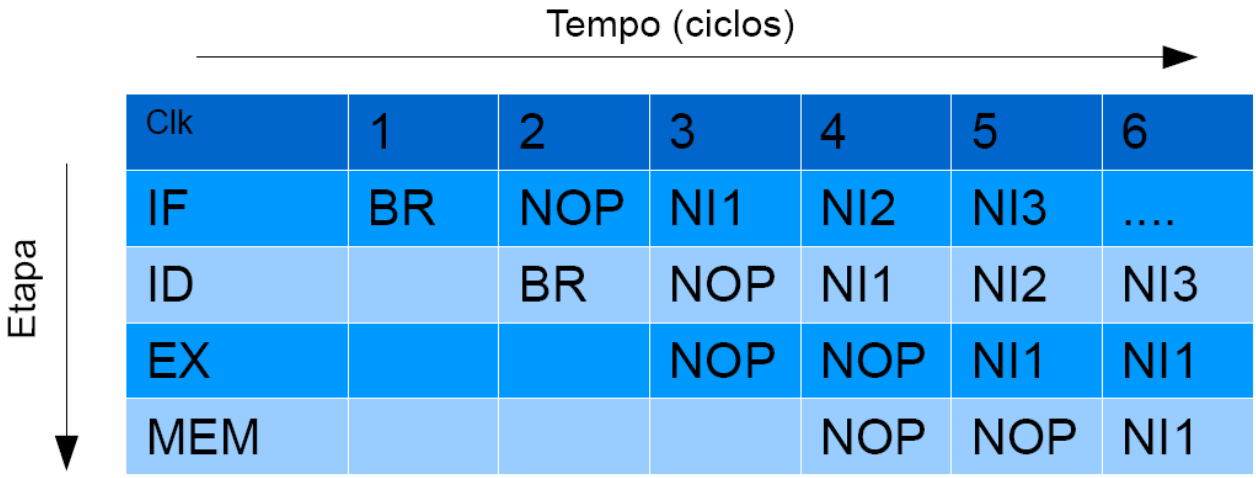


Figura 10: Penalización por unha instrución de salto no esquema modificado

4- Control do Pipeline

O control dun pipeline é altamente dependente da súa arquitectura. Nesta sección presentaremos o concepto dun xeito intuitivo e no contexto do pipeline simple que estudamos nas seccións anteriores. Vamos a estudar dous casos específicos: o caso dun "interlock" (que poderíamos traducilo por "interbloqueo de instrucións"), é dicir, detección de conflitos do pipeline por dependencias de datos, e o control do "bypassing" (é dicir, controlar cando para unha instrución é necesario seleccionar unha entrada de "bypassing" nunha etapa determinada do pipeline).

Interbloqueo de Instrucións

O control necesario debe a) detectar a situación de interbloqueo, e b) parar a emisión de instrucións no pipeline en caso de detección positiva. No pipeline exemplo das seccións anteriores, as situacións de interbloqueo de instrucións detéctanse na etapa ID, xa que ahí é posible coñecer os rexistros que serán os operandos de entrada para a execución da instrución. Neste punto é necesario detectar se nas seguintes etapas do pipeline hai unha instrución que produce un resultado (escritura nun rexistro) que corresponde a un dos rexistros de entrada da instrución na etapa ID. Non se terán en conta as instrucións das seguintes etapas do pipeline que poidan "pasar" ("bypassing" ou "forwarding") o resultado xusto no ciclo en que a instrución da etapa ID o necesite. A detección depende dos tipos de instrucións involucradas. Vexamos un exemplo específico.

LD R1, 45(R2)

DADD R5, R1, R7

DSUB R8, R6, R7

OR R9,R6, R7

Existe unha dependencia entre a instrución de lectura de memoria (LD) e a de suma (DADD) a través do rexistro R1 (cárgase en R1 o contido do enderezo de memoria dado pola suma do contido

de R2 máis un desprazamento de 45, e logo súmase ese valor co contido do rexistro R7 para almacenar o resultado no rexistro R5). Cando DADD se decodifica en ID, a instrución LD está na etapa EX, e non terá listo o resultado que debe cargar en R1.

A Figura 11 amosa un esquema de alto nivel para a detección da situación de interbloqueo para diferentes combinacións de tipos de instrucións (non estas todas as combinacións). O sinal de parada no pipeline debe activarse se a instrución é de tipo ALU ou de salto (BR) ou unha escritura en memoria (ST) e na etapa EX está unha instución LD (load) que escribirá un dos rexistros de entrada da instrución. Debe incluirse tamén algún mecanismo para facer que logo dunha instrución de salto se emita unha NOP (segundo a implementación específica amosada na Figura 9).

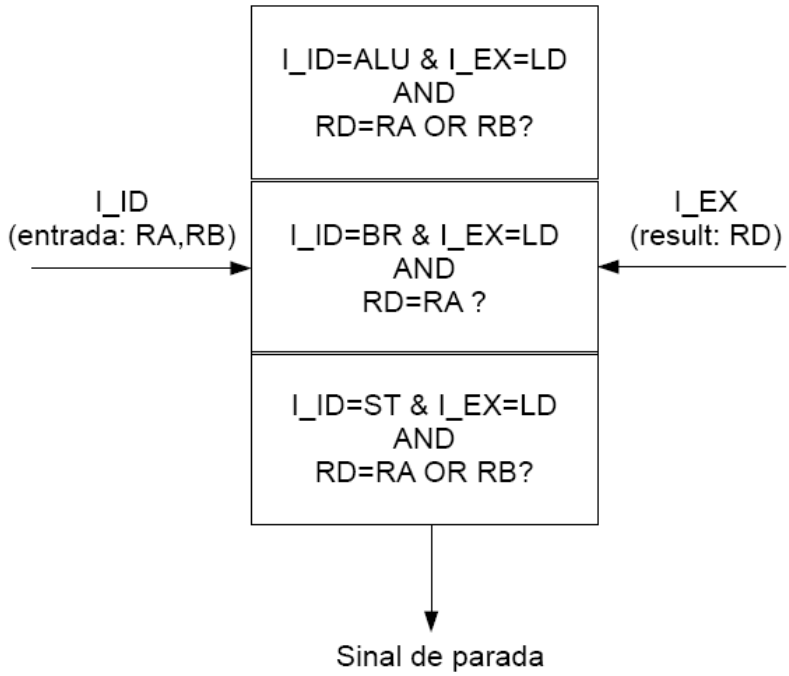


Figura 11: Esquema de alto nivel para control de parada..

Se o sinal de parada está activo na etapa ID, entón no seguinte ciclo de reloxo cargaráse no rexistro da etapa EX un código de non operación (NOP), e os rexistros de pipeline nas etapas ID, e o PC (contador de programa) manteñen os seus contidos (non cargan o que teñen na entrada, e manteñen o valor que tiñan almacenado no ciclo anterior). Cando a condición de parada desaparece, o sinal de parada desactívase e permite a evolución normal das instrucións no pipeline.

Bypassing

O procedemento de control é semellante ao anterior. Require dúas fases: i) detectar se un dos datos pode proceder do bypassing dende outra etapa do pipeline, ii) xerar sinais de control nos multiplexos para seleccionar a procedencia correcta do operando.

A detección faise inspeccionando as diferentes etapas do pipeline para detectar as instrucións que se están executando. A Figura 12 amosa o esquema de control para o bypass na etapa de execución (EX). No control entran os códigos das instrucións nas etapas EX (a que se está executando), MEM e WB (por se poden producir resultados que se pasen por bypass a EX), e prodúcense os sinais de control para os dous multiplexos que controlan os operandos de entrada na ALU.

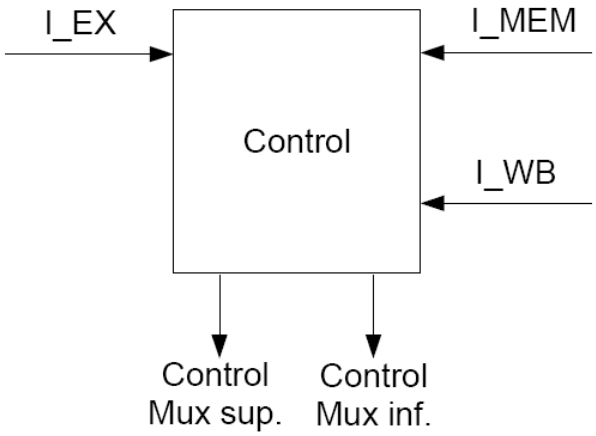


Figura 12: Control do "bypassing" na etapa EX.

O seguinte código ilustra a ventaxa de utilizar “bypassing”:

LD R1, 45(R2)
DADD R5, R6, R7
DSUB R8, R5, R1

A Figura 13 ilustra como debido ao “bypassing” a instrución DSUB non sufre ningunha penalización (en ciclos de espera), xa que as instrucións precedentes que calculan o contido dos rexistros de entrada R1 e R5 traspasan (indicado coas frechas) eses valores directamente dende as etapas MEM (a instrución DADD) e WB (a instrución LD).

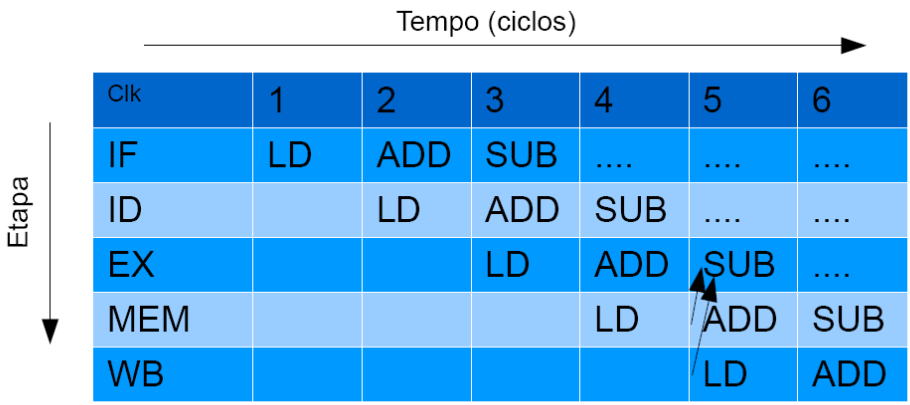


Figura 13: Execución sen paradas debido ao uso do "bypassing".

5- Reducción da Penalización por Saltos

Os saltos constitúen un dos principales problemas para a eficiencia dos pipelines. Nesta sección mostraremos unha serie de estratexias básicas para tratar os saltos. No seguinte tema estudaremos técnicas máis sofisticadas para unha maior eficiencia. As diferentes estratexias para tratar os saltos son as seguintes:

1- Paralizar a obtención de novas instrucións na etapa IF até que se resolva a condición e o destino do salto (en realidade emítense NOP). Neste caso, sempre que se atopa un salto, este leva asociada unha penalización en ciclos de parada (esta foi a estratexia que utilizamos nas seccións anteriores).

2- Predecir sempre o salto coma non tomado: neste caso continúase coa obtención da seguinte instrución secuencial. Debe haber un mecanismo para asegurar que as instrucións posteriores ao salto non cambian o estado da máquina (rexistros e memoria) até que se resolva definitivamente o salto. No pipeline exemplo é doado, xa que as instrucións non modifican o estado (escritura en memoria ou nos rexistros) antes da resolución do salto. Se o salto, finalmente, debe ser tomado, cambiase o código das instrucións presentes no pipeline posteriores ao salto a NOP, e na etapa IF utilízase o enderezo do destino do salto para obter a seguinte instrución. O seguinte exemplo ilustra as dúas situacións que se poden dar:

```
i       BR target
i+1     DADD...
i+2     ....
.....
target  LD.....
```

Na Figura 14(a) amósase o caso para salto tomado e na Figura 14(b) o caso para salto non tomado.

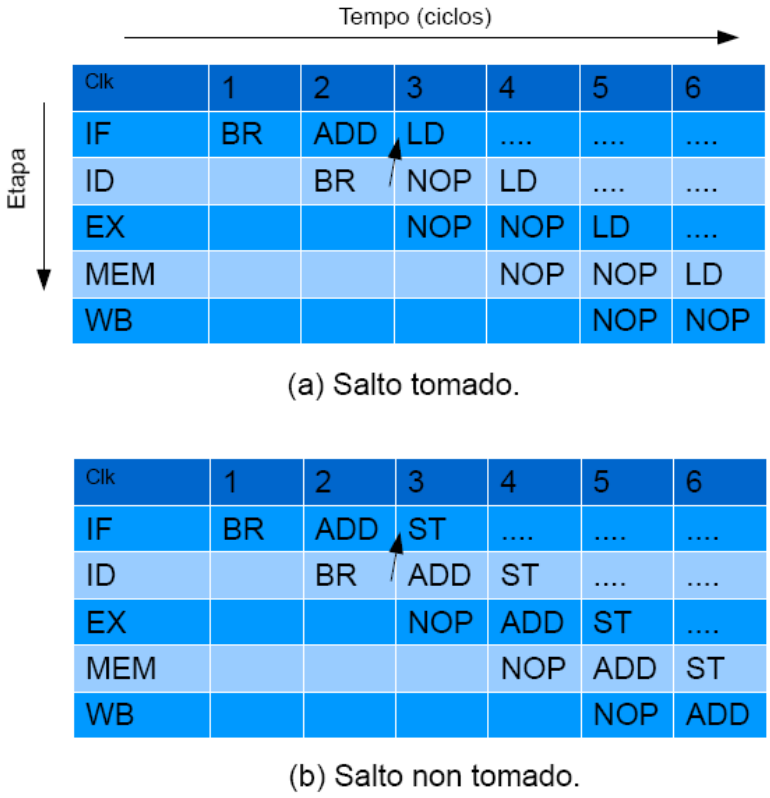


Figura 14: Exemplo do efecto da estratexia "salto sempre non tomado".

3- Predecir sempre o salto coma tomado: non ten ventaxas se o enderezo de destino do salto non é coñecido antes que a resolución do salto.

4- Salto retardado (“delayed branch” en inglés):O compilador colocará logo da instrución de salto certo número de instrucións (tantas coma sexa necesario até que se resolva o salto e se coñeza o destino do salto) que permitan facer traballo (executar instrucións) que probablemente resulte útil. Polo tanto, logo dun salto, nunca parará a execución, e pode ser necesario converter estas instrucións en NOP antes de que cambien o estado (se a transformación non é compatible co resultado real do salto durante a execución).

A Figura 15 amosa as diferentes transformacións que o compilador pode levar a cabo no código para o caso dun instrución especulativa. Logo da instrución de salto existe un ciclo no que non se coñece o resultado do salto nin o seu destino. Polo tanto nese ciclo existen varias alternativas para seleccionar a instrución a executar. Este ciclo corresponde a instrución inmediatamente posterior ao salto no código. Na opción (a) execútase unha instrución independente do salto que se atopada antes deste na orde do programa. Na opción (b), a instrución anterior determina o resultado do salto, polo que no se pode mover á posición posterior. Neste caso é a instrución de destino do salto a que se copia na posición inmediatamente posterior ao salto. Notar que no programa a instrución permanece tamén na súa posición orixinal, xa que pode ser destino doutros saltos, e o enderezo do salto increméntase unha posición. Na opción (c) seleccionase a instrución inmediatamente posterior ao salto. A opción seleccionada dependerá da información dispoñible durante a compilación, e en todo caso debe asegurarse que a execución da instrución non cambia o estado antes de coñecerse o resultado da instrución de salto.

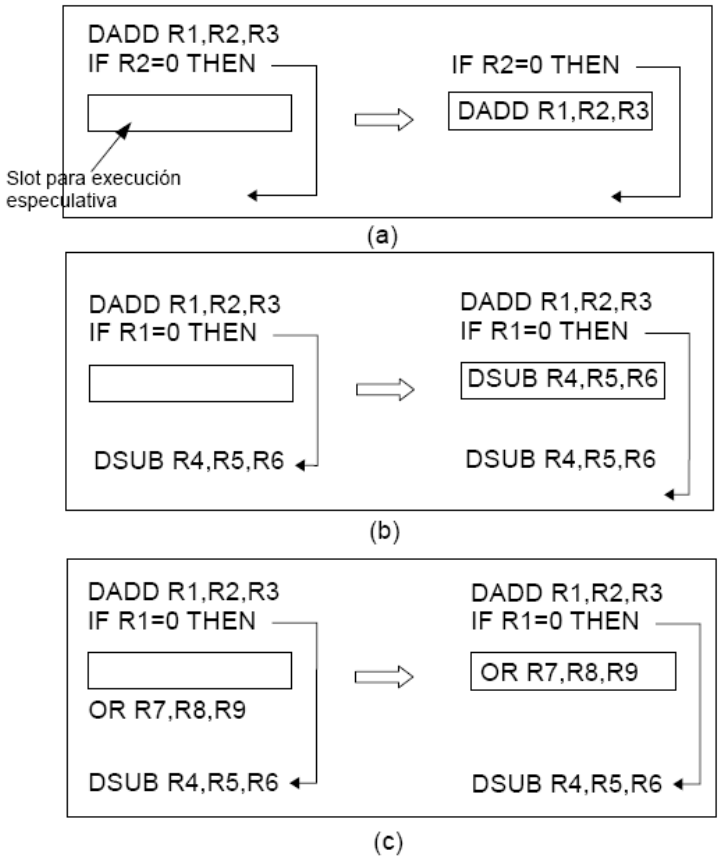


Figura 15: Transformacións de código para salto retardado.

6- Extensión a Operacións Multiciclo e Varias Unidades

Nesta sección discutimos brevemente como extender o pipeline exemplo para operación máis complexas, coma operación de punto flotante ou multiplicación e división de números enteiros. O pipeline resultante amósase na Figura 16 (este é un diagrama de alto nivel onde se omiten moitos dos detalles). As modificacións máis salientábeis son as seguintes:

- Engádese un banco de rexistros para enteiros e outra para números de punto flotante.
- Engadimos dúas unidades de execución, segmentadas en varias etapas (ciclos) para multiplicación de enteiros e de punto flotante (segmentada en 7 etapas), e suma de punto flotante (segmentada en 4 etapas).
- Engádese unha unidade de división non segmentada que require varios ciclos (por exemplo 25 ciclos) reutilizando a mesma etapa para completar a execución (nesta unidade non se pode executar unha instrución por ciclo).

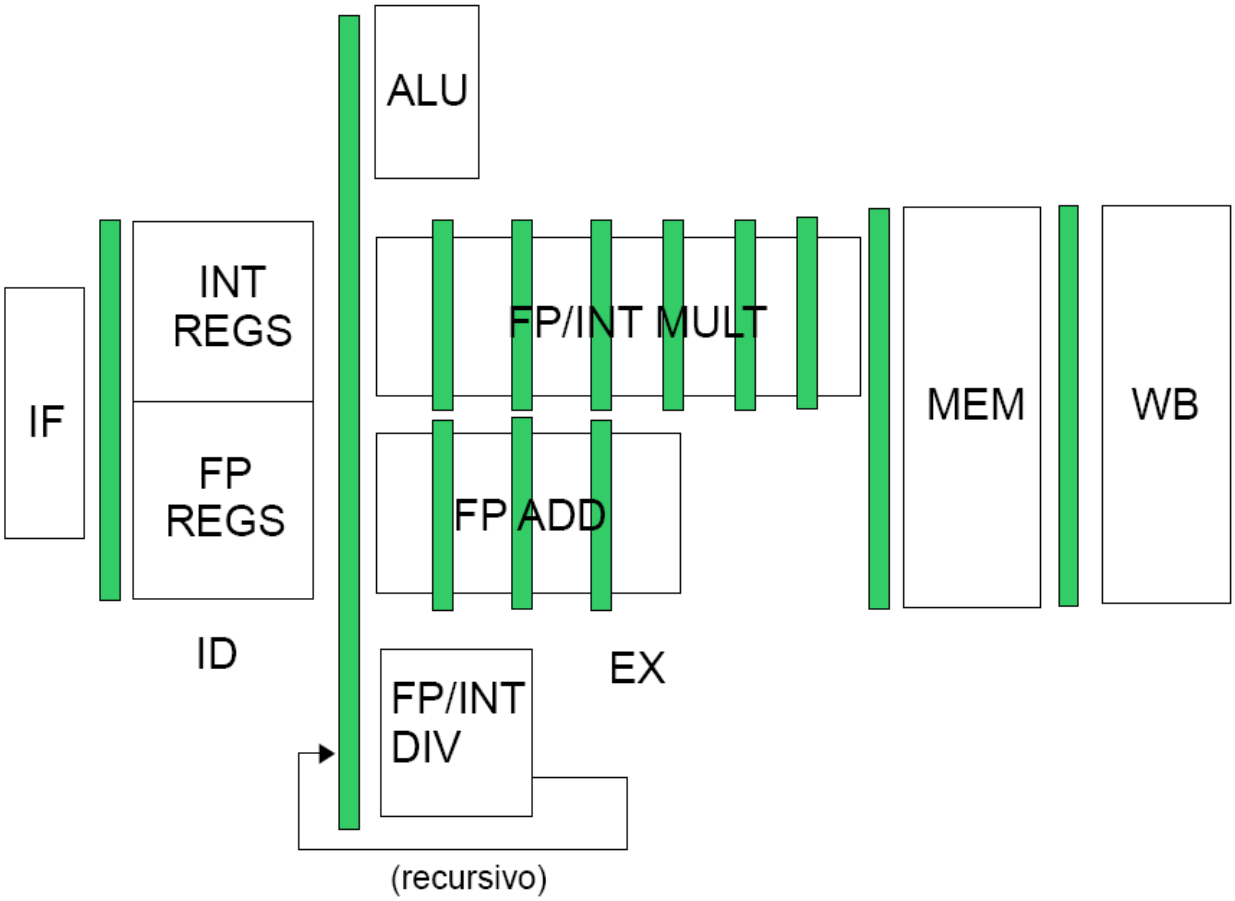


Figura 16: Pipeline con operacións multiciclo e múltiples unidades de execución.

Dun xeito breve, a detección de conflitos neste pipeline é da seguinte forma (asumimos que a detección de conflictos faise na etapa ID) :

Chequeo de conflitos estruturais: é necesario detectar os conflitos na unidade de execución de divisións, e na etapa WB pode haber conflitos na escritura de rexistros, dependendo do número de portos dispoñibles para escritura no banco de rexistros.

Unha estratexia sinxela para detectar estes conflictos e a seguinte:

- Parar a emisión da instrución a execución se é unha división e outra división se está a executar.
- Parar a emisión se se detecta un posible conficto na etapa WB (número de instrucións que coinciden simultáneamente na etapa WB maior que o número de portos de escritura dos bancos de rexistros). A detección require comparar o número de ciclos que tardará en chegar á etapa WB a instrución actual (depende do tipo de instrución) co número de ciclos que tardaran en chegar as instrucións que se están executando no pipeline.

Chequeo de conflitos por dependencias (coñecidos coma conflitos RAW, por lectura logo de escritura – Read After a Write en inglés): este conflitos prodúcense cando unha instrución xa en execución escribirá un dos rexistros que ten que ler a instrución que se pretende pasar á etapa de execución. A detección é semellante ao do pipeline sinxelo, só que aparecen máis casos posibles. Unha instrución será lanzada a execución se os rexistros que ten que ler non aparecen como rexistros destino en etapas do pipeline dende as que non é posible que o resultado estea dispoñible para ser usado pola instrución, ben por lectura directa do rexistro na etapa ID ou por bypassing.

Chequeo de conflitos tipo WAW (escritura logo de escritura, Write After a Write en inglés): é necesario determinar se hai algunha instrución no pipeline (logo da etapa ID), que ten como destino o mesmo rexistro que a instrución na etapa ID, pero que faría a escritura do rexistro un número de ciclos posterior (por exemplo se a instrución no pipeline é unha división que acaba de comenzar, e a instrución na etapa ID é unha suma de punto flotante). Se este é o caso, entón parar a emisión á etapa de execución (as escrituras deben producirse en orde). Existen outros mecanismos máis sofisticados, pero a frecuencia de ocorrencia deste tipo de conflito é moi baixa, así que este mecanismo simple é suficiente.

Para o caso do "bypassing", os conceptos son os mesmos que para o pipeline simple. É necesario detectar se para unha instrución, en cada etapa, debe seleccionarse una camiño de "bypass" para obter un dato de entrada.

7- Excepcións e Pipelining

Un problema adicional importante cando se deseña un pipeline é o das excepcións. As excepcións poden producirse por varios eventos (operacións de punto flotante, fallos de páxina, etc). Os casos complexos son os que requiren que o programa volva a executarse dende o punto que produciu a excepción, logo de tratar a mesma (cunha rutina software especial). A excepción pode producila unha instrución cando xa está nunha etapa avanzada do pipeline, o cal implica que outras instrucións posteriores segundo a orde do programa, xa entraron no pipeline. Para executar as instrucións dende a que produciu a excepción, é necesario conservar o estado da máquina nese punto preciso (contido de rexistros e memoria). As excepcións que permiten isto chámanse "excepcións precisas".

Vexamos un exemplo para ilustrar a problemática da situación:

DIV.D F0,F2,F4 (División en dobre precisión)

ADD.D F10,F10,F8 (Suma en dobre precisión)

SUB.D F12, F12, F14 (Resta en dobre precisión)

Estas tres instrucións son independentes, e poden executarse sen paradas no pipeline. Tomando como exemplo o pipeline da sección anterior, malia que DIV.D está primeiro segundo a orde do programa, rematará (escribirá o resultado no banco de rexistros de punto flotante) logo de ADD.D e

SUB.D, é dicir teremos unha terminación forma de orde, debido á diferencia do número de ciclos para execución de cada instrución. O problema coa terminación fora de orde son as excepcións. Pode ocorrer unha excpeción en DIV.D (por exemplo un overflow) cando xa ADD.D e SUB.D escribiran os seus resultados no banco de rexistros, polo tanto cambiando o estado do punto no que se produciu a excepción. Os procesadores engaden soporte para tratar excepción exactas, pero isto implica reducir as prestacións. Algunhas técnicas que se teñen implementado son as seguintes:

1- Incorporar outro banco de rexistros no que se gardan os novos valores producidos por cada instrución. Os novos valores cópianse ao banco de rexistros orixinal cando todas as instrucións anteriores á que produciu o resultado xa se completaron (sen producir excepcións). Se se produce unha excepción, o banco de rexistros principal garda o estado orixinal anterior á excepción.

2- Outra opción é utilizar unha rutina software, de tal xeito que sempre se teña suficiente información para reproducir a secuencia de instrucións se se produce unha excepción. Por exemplo na seguinte secuencia:

I1 -> instrución que tarda moitos ciclos e que presenta unha excepción.

I2

... instrucións emitidas pero sen completar.

In-1

In -> instrución emitida e rematada.

In+1..

A rutina debería se quen de simular a nova execución de I1 até In-1, sen a excepción e logo seguir coa execución no pipeline na instrución n+1.

3- Outra opción é permitir emitir instrucións se se pode asegurar que as instrucións emitidas con anterioridade non causarán unha excepción. Isto obriga a parar a emisión no pipeline en caso de risco de excepción. Isto require hardware para a detección do risco de excepción na etapa EX, ao principio, para parar a emisión de instrucións posteriores.

8- Exemplo de Pipeline Realista (sinxelo): MIPS R4000

Nesta sección describimos unha implementación real dun procesador sinxelo, o MIPS R4000. No tema seguinte veremos exemplos máis avanzados incorporando execución superscalar e fora de orde.

O MIPS R4000 incorpora un total de 8 etapas de pipeline. As instrucións son lidas da cache de instrucións e os datos lidos/escritos na cache de datos. A diferencia fundamental co sistema que estudamos con cinco etapas de pipeline, é a segmentación dos accesos ás caches en varias etapas.

A Figura 17 amosa as diferentes etapas neste pipeline. As diferentes etapas son as seguintes:

IF: parte inicial da lectura da instrución. Seleccionase o enderezo da seguinte instrución e comenza a lectura da cache de instrucións.

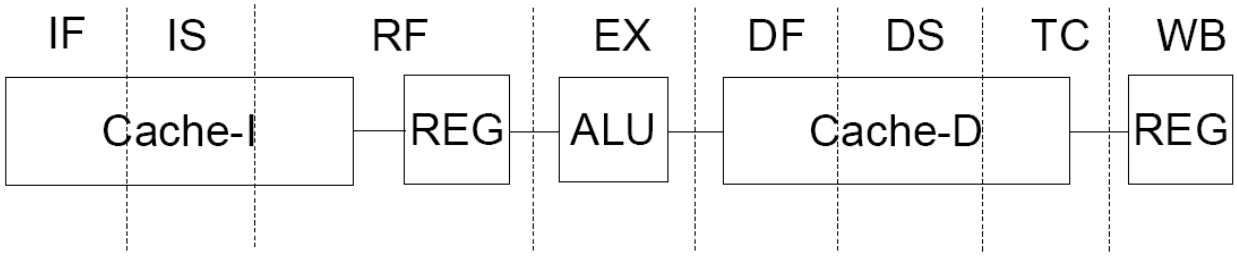


Figura 17: Esquema de alto nivel do pipeline do procesador MIPS R4000.

IS: segunda parte da lectura da instrución, na que se completa a lectura da cache de instrucións.

RF: chequeo da etiqueta (tag) do enderezo da liña cache para comprobar que a lectura corresponde a unha liña correcta. Logo faise a decodificación da instrución, lectura de rexistros e chequeo de conflitos da instrución.

EX: execución da instrución (cálculo de enderezos, operación ALU, cálculo do destino de saltos e avaliación de condicións para saltos).

DF: fase inicial da lectura/escritura de datos da cache.

DS: segundo ciclo de lectura/escritura de datos, completando o acceso á cache de datos.

TC: chequeo da etiqueta do enderezo da liña cache para determinar se a lectura/escritura foi un acerto.

WB: etapa de escritura ao banco de rexistrs para LOAD e operacións rexistro a rexistro (por exemplo operacións ALU, etc).

Esta versión do pipeline presenta as seguintes diferencias en comparación co pipeline exemplo:

- Máis camiños de "bypassing", por exemplo dende DF, DS, TC e WB á EX.
- Máis latencia para instrucións dependentes dun load.
- Máis latencia na resolución de saltos.

A Figura 18 amosa un exemplo para instrucións dependentes dun load (cada columna da táboa indica a etapa do pipeline que executa unha parte da correspondente instrución). Neste pipeline incórrese en dous ciclos de parada (Stall) debido á dependencia (a instrución DADD queda parada na etapa IS, e impide avanzar as instrucións posteriores na orde do programa). Logo da segunda etapa de acceso á cache de datos, faise un bypass do dato á etapa de execución. Se DADD e DSUB non dependesen de LD, entón a instrución OR tería que recoller o dato de entrada de R1 a través de bypassing.

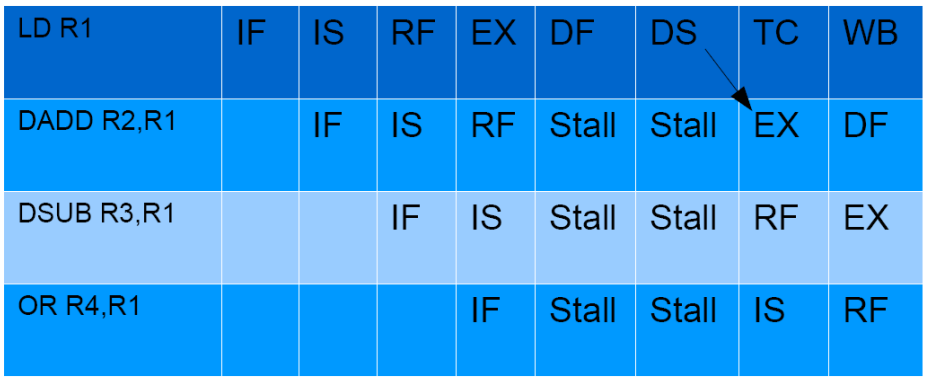


Figura 18: Ilustración da parada por dependenza dun load.

Respecto das instrucións de salto, nesta arquitectura transcorren tres ciclos dende o inicio da instrución até que se coñece o destino do salto. (computado no ciclo EX). Un dos ciclos serve de "slot" para un salto retardado, intentando incorporar unha instrución no primeiro ciclo logo da instrución de salto, segundo as técnicas estudadas anteriormente. Nos outros dous ciclos séguese a estratexia de predicir o salto coma non tomado. Na Figura 19 amósase as dúas situacións que poden darse logo dunha instrución de salto, dependendo de se o salto finalmente é tomado ou non tomado. No caso de salto non tomado, o slot logo da instrución de salto é seleccionado polo compilador en función da información dispoñible en tempo de compilación. Nos dous seguintes ciclos séguese a estratexia de predicir o salto coma non tomado, co que se emiten as instrucións que seguen en secuencia a instrución de salto. A única precaución neste caso é se a instrución do "slot" foi tomada coherentemente co resultado final do salto. Se esta instrución non é coherente debe anularse no pipeline antes de que cambie o estado do sistema. No caso de salto tomado, o slot é tamén selecconado polo compilador e esa instrución debe ser coherente co resultado final do salto. Para os outros dous ciclos, como a predición é salto non tomado, cando se resolve o salto na etapa EX, deben anularse as instrucións emitidas (cambialas a código NOP).

BR	IF	IS	RF	EX	DF	DS	TC	WB
Slot		IF	IS	RF	EX	DF	DS	TC
IS+2			IF	IS	RF	EX	DF	DS
IS+3				IF	IS	RF	EX	DF
IS+4					IF	IS	RF	EX

(a) Salto non tomado.

BR	IF	IS	RF	EX	DF	DS	TC	WB
Slot		IF	IS	RF	EX	DF	DS	TC
IS+2			IF	IS	NOP	NOP	NOP	NOP
IS+3				IF	NOP	NOP	NOP	NOP
Destino Salto					IF	IS	RF	EX

(b) Salto tomado.

Figura 19: Execución no pipeline para (a) salto non tomado e (b) salto tomado.

Nunha avaliación deste pipeline para algúns programas de proba con cargas de traballo de números enteiros (unha versión antiga dos programas SPECint) o CPI medio acadado é de 1.54 (con valores entre 1.20 e 1.88, dependendo do programa), coas contribucións por dependencia de loads ao CPI de 0.16 (rango entre 0.13-0.27), e por penalización de saltos unha contribución ao CPI de 0.38 (rango entre 0.06-0.61). Obviamente o CPI ideal que podería acadar esta arquitectura é de 1.0 (rematar unha instrución por ciclo).