MICROARQUITECTURA

1) Definir el concepto de ISA vs Microarquitectura. ¿Para qué sirve esta distinción? - ¿Pueden haber dos procesadores con la misma ISA y distinta microarquitectura? Dar ejemplo. - ¿Pueden haber dos procesadores con la misma microarquitectura y distinta ISA? Dar ejemplo.

```
La Instruction Set Architecture es una especificación que provee una interfaz
del procesador, para ser utilizada por el programador, el compilador, el
sistema operativo, etcétera. Dicha interfaz establece el conjunto de
instrucciones soportado por esta arquitectura, la forma de organización y
direccionamiento de la memoria, los privilegios de ejecución, los registros,
así como cualquier otro detalle relevante a la hora de implementar algo que
funcione sobre un procesador que cumpla con esta interfaz.

La microarquitectura, en cambio, es una implementación particular de la ISA.
Incluye, a grandes rasgos, la organización y detalles específicos de los
componentes internos del procesador (caché, buses, unidades de ejecución,
etc).

Esta distinción permite que se generen familias de procesadores de una misma
arquitectura, que aseguren la compatibilidad entre distintos modelos. Esto
significa que un código corriendo en un modelo de procesador de una
determinada arquitectura debería de poder correr un procesador de esa misma
arquitectura, pero de otro modelo. Sin contar con esta ventaja, un binario
compilado para una determinada arquitectura, debería ser recompilado cada vez
que se quisiera usar en un procesador distinto.

Un ejemplo puntual de procesadores con la misma ISA y distintas
microarquitecturas se da en la arquitectura x86, en donde se tiene por un lado
al modelo Intel Pentium 4, que está implementado con microarquitectura
Netburst, propietaria de Intel, y al AMD Athlon, que está implementado con una
microarquitectura distinta, propietaria de AMD.

Dado que la microarquitectura es una implementación particular de la ISA, no
tiene sentido hablar de dos procesadores con la misma microarquitectura y
distinta ISA.
```

2) ¿Dentro de una microarquitectura, cuál es la diferencia entre la organización y el hardware? - ¿Pueden haber dos procesadores con la misma organización y distinto hardware? Dar ejemplo. - ¿Pueden haber dos procesadores con el mismo hardware y distinta organización? Dar ejemplo.

```
La organización es todo lo que respecta a los distintos componentes que
conforman la microarquitectura, cómo funcionan y cómo se relacionan entre sí
los distintos bloques funcionales internos de la CPU. El hardware son detalles
implementativos, de nivel lógico, electrónico y de fabricación.

Es posible encontrar procesadores con la misma ISA y la misma organización,
pero distinto hardware, por ejemplo un procesador orientado a uso de
escritorio, vs el mismo modelo de procesador pero orientado a su uso en
dispositivos móviles. Se puede considerar, por ejemplo, dentro de la
```

```
microarquitectura Netburst, a los procesadores Pentium 4 vs los procesadores
Pentium 4-M. Siendo muy similares en cuanto a los detalles
microarquitecturales, pero con diferencias importantes en cuanto a su diseño
electrónico, por ejemplo incluyendo el segundo una mayor optimización y lógica
adicional de control de consumo energético.
```

INSTRUCTION LEVEL PARALELISM

1) Distinguir entre microarquitecturas monociclo vs multiciclo, mencionando ventajas y desventajas de cada tipo.

```
Las microarquitecturas monociclo son aquellas en donde cada instrucción es
ejecutada en un ciclo de máquina, y en donde se considera que no existen
cambios intermedios dentro del contexto de la ejecución de una misma
instrucción. Es decir, si la instrucción transforma el estado arquitectural AS
en AS', entonces la trancisión AS -> AS' se realiza de forma atómica, en un
ciclo de clock, sea cual sea la instrucción que se esté ejecutando. Es una
arquitectura muy simple. A nivel electrónico, se puede representar como una
máquina de estados con dos bloques, uno siendo la lógica secuencial, que
preserva el estado de la CPU, y el otro la lógica combinatoria, que resuelve
la ejecución de la instrucción. Presenta como desventaja que la duración de un
ciclo generalmente es larga, ya que está supeditada al tiempo que tarde en
ejecutarse la más lenta de las instrucciones definidas por el ISA,
considerando el peor caso posible. Ejemplo, una instrucción de punto flotante,
que además realiza acceso a memoria. Esto significa que hay mucho "tiempo
desperdiciado", ya que aquellas instrucciones simples y cuya ejecución podría
realizarse en mucho menor tiempo, ejemplo incrementar un registro, se ejecutan
en la primera parte de ese ciclo, y el resto es tiempo muerto.

Una posible mejora sobre esto último, es una arquitectura multiciclo, en donde
se divide la ejecución en varias etapas, y cada una de ellas corre en un ciclo
de máquina. De forma similar, la duración de un ciclo de máquina está
supeditada al tiempo que pueda tardar la más lenta de las etapas. Esto permite
paralelizar la ejecución de instrucciones, por ejemplo a través de un
pipeline, ya que si la cantidad de etapas es N, en el caso ideal se pueden
tener N instrucciones corriendo al mismo tiempo (una sobre cada etapa).

Finalmente, existen las arquitecturas basadas en microcódigo, en donde el
tiempo de clock puede ser muy pequeño, y cada etapa está subdivida en
microinstrucciones. La duración de las etapas no siempre es la misma, ya que
depende de las microinstrucciones que requiera la ejecución de cada
instrucción. Esto permite optimizar la ejecución a muy bajo nivel, por ejemplo
aplicando técnicas como Out-Of-Order Execution.
```

2) Explicar qué es el pipelining, y cómo ayuda a mejorar el rendimiento del procesador. - Diagramar un ejemplo de un flujo de ejecución sin pipelining vs. la misma ejecución con pipelining.

```
El pipelining es una técnica que permite paralelizar la ejecución de
instrucciones, utilizando para ello las distintas etapas del procesador, de
```

forma similar a una cinta de producción en una fábrica, o en una tubería, en donde el líquido ingresa de un lado, y sale del otro; en este último ejemplo la tubería inicialmente tarda un tiempo en llenarse (imaginemos que es una tubería de cientos de metros), pero una vez que lo hace se establece un flujo de transmisión contínuo. Lo mismo sucede con una arquitectura con pipeline, en donde se busca generar un flujo de instrucciones constante. De este modo, en un pipeline ideal, el tiempo por instrucción resulta inversamente proporcional a la cantidad de etapas del pipeline.

Supongamos el flujo de ejecución de las instruciones A, B, C.

Sin pipelining:

Ciclo	Fetch	Decode	Execute	Retire
1	A			
2		A		
3			A	
4				A
5	B			
6		B		
7			B	
8				B
9	C			
10		C		
11			C	
12				C
13	D			
14		D		
15			D	
16				D

Con pipelining:

Ciclo	Fetch	Decode	Execute	Retire
1	A			
2	B	A		

```
-------------------------------------------------------------
    3   |     C     |     B     |     A         |
-------------------------------------------------------------
    4   |     D     |     C     |     B         |      A
-------------------------------------------------------------
    5   |           |     D     |     C         |      B
-------------------------------------------------------------
    6   |           |           |     D         |      C
-------------------------------------------------------------
    7   |           |           |               |      D
-------------------------------------------------------------

Como se puede ver, hay una mejora notable de rendimiento. En el ciclo 4 se
alcanza una condición de estabilidad del pipeline, en donde todas las etapas
se encuentran haciendo algo al mismo tiempo. En este contexto, si se mantienen
las condiciones óptimas, y continúan ingresando instrucciones al pipeline de
forma ininterrumpida, el throughput se maximiza, ya que el TPI se divide según
la cantidad de etapas del pipeline.
```

3) ¿Qué son los obstáculos/riesgos, y qué fenómeno pueden ocasionar en relación al pipeline? - Diagramar un ejemplo sencillo de un flujo de ejecución en donde se pueda observar este fenómeno.

```
Los obstáculos son dependencias entre etapas del pipeline, que hacen que una o
más etapas no puedan ejecutarse. Esto genera un fenómeno denominado pipeline
stall, que consiste en la detención parcial o total de las distintas etapas
del pipeline. Dado que cada etapa debe ejecutarse, y pasar la instrucción a la
siguiente etapa, las etapas anteriores a la que no puede ejecutarse terminan
deteniéndose también, al generarse una reacción en cadena en la que las
instrucciones no pueden avanzar de etapa. Las etapas posteriores, por otro
lado, siguen trabajando con las instrucciones que ya tenían en ejecución, pero
al no recibir más instrucciones, se van deteniendo.


Pipeline con stall de 5 ciclos en la instrucción E en la etapa 2.

-------------------------------------------------
 Ciclo  |  E1  |  E2  |  E3  |  E4  |  E5  |
-------------------------------------------------
   1    |  A   |      |      |      |      | <- LLENANDO
-------------------------------------------------
   2    |  B   |  A   |      |      |      | <- LLENANDO
-------------------------------------------------
   3    |  C   |  B   |  A   |      |      | <- LLENANDO
-------------------------------------------------
   4    |  D   |  C   |  B   |  A   |      | <- LLENANDO
-------------------------------------------------
   5    |  E   |  D   |  C   |  B   |  A   | <- ESTABLE
-------------------------------------------------
   6    |  F   |  E   |  D   |  C   |  B   | <- ESTABLE
-------------------------------------------------
   7    |  F   |  E   |  -   |  D   |  C   | <- STALL
-------------------------------------------------
   8    |  F   |  E   |  -   |  -   |  D   | <- STALL
```

```
-------------------------------------------------
   9    |   F   |   E   |   -   |   -   |   -   | <- STALL
-------------------------------------------------
   10   |   F   |   E   |   -   |   -   |   -   | <- STALL
-------------------------------------------------
   11   |   F   |   E   |   -   |   -   |   -   | <- STALL
-------------------------------------------------
   12   |   G   |   F   |   E   |   -   |   -   | <- RECUPERANDO
-------------------------------------------------
   13   |   H   |   G   |   F   |   E   |   -   | <- RECUPERANDO
-------------------------------------------------
   14   |   I   |   H   |   G   |   F   |   E   | <- ESTABLE
-------------------------------------------------


El mismo pipeline sin stall.

-------------------------------------------------
 Ciclo  |   E1  |   E2  |   E3  |   E4  |   E5  |
-------------------------------------------------
   1    |   A   |       |       |       |       | <- LLENANDO
-------------------------------------------------
   2    |   B   |   A   |       |       |       | <- LLENANDO
-------------------------------------------------
   3    |   C   |   B   |   A   |       |       | <- LLENANDO
-------------------------------------------------
   4    |   D   |   C   |   B   |   A   |       | <- LLENANDO
-------------------------------------------------
   5    |   E   |   D   |   C   |   B   |   A   | <- ESTABLE
-------------------------------------------------
   6    |   F   |   E   |   D   |   C   |   B   | <- ESTABLE
-------------------------------------------------
   7    |   G   |   F   |   E   |   D   |   C   | <- ESTABLE
-------------------------------------------------
   8    |   H   |   G   |   F   |   E   |   D   | <- ESTABLE
-------------------------------------------------
   9    |   I   |   H   |   G   |   F   |   E   | <- ESTABLE
-------------------------------------------------
   10   |   J   |   I   |   H   |   G   |   F   | <- ESTABLE
-------------------------------------------------
   11   |   K   |   J   |   I   |   H   |   G   | <- ESTABLE
-------------------------------------------------
   12   |   L   |   K   |   J   |   I   |   H   | <- ESTABLE
-------------------------------------------------
   13   |   M   |   L   |   K   |   J   |   I   | <- ESTABLE
-------------------------------------------------
   14   |   N   |   M   |   L   |   K   |   J   | <- ESTABLE
-------------------------------------------------
```

4) Explicar qué son los obstáculos estructurales. Dar un ejemplo. ¿De qué modo se pueden prevenir?

```
Los obstáculos estructurales son aquellos que surgen por dependencias entre
etapas que se encuentran en el propio diseño de la microarquitectura. Esto
puede sugerir que las etapas no fueron diseñadas con la suficiente
granularidad o independencia entre sí, aunque muchas veces la solución no es
```

```
tan sencilla como "dividir la etapa en más pequeñas".

Un ejemplo típico es cuando hay una etapa que hace el fetch de las
instrucciones (para ello requiere realizar un acceso a memoria), y una etapa
distinta que hace el fetch de los operandos (para ello requiere también
realizar un acceso a memoria).

Lo anterior se puede ver en un diagrama, suponiendo que la instrucción A
necesita sacar operandos de memoria. Esto hace que el fetch de la instrucción
C se tenga que pausar, generalmente insertando en el pipeline una operación
NOP (también llamado burbuja).

-------------------------------------------------------------------------
 Ciclo  |   Fetch   |   Decode  |   Fetch Oper  |   Execute     |   Retire
-------------------------------------------------------------------------
    1   |     A     |           |               |               |
-------------------------------------------------------------------------
    2   |     B     |     A     |               |               |
-------------------------------------------------------------------------
    3   |     C     |     B     |     A         |               |
-------------------------------------------------------------------------
    4   |     C     |     -     |     B         |      A        |
-------------------------------------------------------------------------
    5   |     D     |     C     |     -         |      B        |     A
-------------------------------------------------------------------------
    6   |     E     |     D     |     C         |      -        |     B
-------------------------------------------------------------------------

La mayoría de las veces prevenir estos obstáculos implica agregar o
reorganizar el hardware. Siguiendo con el ejemplo anterior, una solución sería
desdoblar la caché L1 en caché L1 de instrucciones (usada por el Fetch) y
caché L1 de datos (usada por el Fetch de Operandos).
```

5) Explicar qué son los obstáculos de datos. - Explicar qué tipo de dependencias de datos definen las siglas WAR, RAW, WAW.

```
Las dependencias de datos son aquellas que pueden surgir debido al contexto y
la naturaleza de las instrucciones contenidas en el pipeline. Existen tres
tipos de dependencias:

- Read-After-Write (RAW): También llamada dependencia de flujo, o dependencia
  real. Sucede cuando una instrucción escribe un dato como resultado, y una
  instrucción posterior necesita utilizar ese mismo dato como entrada. Si la
  dependencia no es detectada, la instrucción posterior termina leyendo un dato
  inválido. Cuando el riesgo es detectado, si la instrucción que provee el dato
  tarda mucho en ejecutarse, esto termina ocasionando un pipeline stall.

  Supongamos que la instrucción A produce como salida el el dato que la
  instrucción B va a usar como operando.

  En un flujo de ejecución normal (sin detección de dependencias), termina
  sucediendo que en el ciclo 4 la instrucción B termina leyendo un operando
  que contiene información inconsistente con el orden de ejecución del
  programa. Es decir, a partir de este punto, el programa no sirve más.
```

Ciclo	Fetch	Decode	Fetch Oper	Execute	Retire
1	A				
2	B	A			
3	C	B	A		
4	D	C	B	A	
5	E	D	C	B	A
6	F	E	D	C	B

Si agregamos detección de dependencias, la solución es detener el fetch de los operandos de B, hasta que A haya guardado el resultado en el registro o memoria correspondiente.

Ciclo	Fetch	Decode	Fetch Oper	Execute	Retire
1	A				
2	B	A			
3	C	B	A		
4	D	C	B	A	
5	D	C	B	-	A
6	D	C	B	-	-
7	E	D	C	B	-
8	F	E	D	C	B

- Write-After-Read (WAR): También llamada anti-dependencia. Sucede cuando una instrucción lee un dato, y luego otra escribe ese mismo dato. El riesgo es que la instrucción que escribe se termine ejecutando antes de la instrucción que lee, afectando la consistencia del programa. Esto puede llegar a suceder en un contexto de Ejecución Fuera de Orden.

- Write-After-Write (WAW): También llamada dependencia de output. Sucede cuando una instrucción escribe un dato, y luego otra instrucción escribe ese mismo dato. En un flujo de ejecución normal, el dato debería quedar con el valor que escribió la segunda instrucción. El riesgo es que la escritura de la segunda instrucción repercuta primero que la de la primera instrucción, por lo que el dato terminaría siendo el que definió la primera instrucción, haciendo que el programa llegue a un estado de inconsistencia. Esto puede llegar a suceder en un contexto de Ejecución Fuera de Orden.

6) Explicar los conceptos de scheduling estático vs scheduling dinámico. - Dar ejemplos de scheduling estático. - ¿Qué es el interlocking, y cómo se relaciona con el scheduling estático?

El scheduling estático son aquellas soluciones que buscan maximizar el ILP, y solucionar los problemas de dependencias, pero que son implementados en nivel del compilador, mientras que en el scheduling dinámico el procesador interviene de forma dinámica para mejorar el ILP.

Ejemplos de scheduling estático pueden ser la inserción de burbujas NOP entre las instrucciones que tengan dependencias de datos, como forma de eliminar estas dependencias en tiempo de compilación, el loop unrolling que consiste en transformar los loops en una secuencia de instrucciones sin saltos, el ordenamiento de instrucciones de un código de forma tal que corran en el procesador de la forma más eficiente posible, etcétera.

El interlocking es una técnica que permite que cuando una determinada etapa se encuentra ocupada, o cuando existe una dependencia de algún tipo, las etapas afectadas puedan detener su ejecución (stall), o insertar burbujas (NOP). Básicamente, es un mecanismo de detección y protección ante dependencias, aunque no sirve para solucionar la pérdida de rendimiento ocasionada. El interlocking es una forma de scheduling dinámico, y requiere contar con lógica adicional, específica para realizar esta tarea. Se relaciona con el scheduling dinámico, ya una posibilidad es minimizar las capacidades de interlocking de un procesador, o directamente no hacer interlocking (como en el caso de los MIPS en sus orígenes), y delegar esta tarea al compilador, que se debe encargar de generar un código apropiado que evite las dependencias (por ejemplo, insertando NOPs entre dos instrucciones conflictivas cuando el reordenamiento no es suficiente).

7) ¿Cómo funciona el forwarding en el contexto de los obstáculos de datos, y para qué sirve? - ¿Qué dependencias de datos se ven afectadas por su utilización?

El forwarding es una técnica que consiste en enviar el resultado de la ejecución de una instrucción, directo desde la unidad de ejecución, hacia la instrucción que lo necesite como dependencia (en el caso de RAW).

Manteniendo el mismo ejemplo que se utilizó en la explicación de RAW, supongamos que la ejecución B requiere como entrada el dato emitido por la ejecución de A.

Con Forwarding

Ciclo	Fetch	Decode	Fetch Oper	Execute	Retire
1	A				
2	B	A			
3	C	B	A		
4	D	C	B	A (->B)	

```
-------------------------------------------------------------------------
    5   |     D     |     C     |     B (<-A)   |      -        |     A
-------------------------------------------------------------------------
    6   |     E     |     D     |     C         |      B        |     -
-------------------------------------------------------------------------
    7   |     F     |     E     |     D         |      C        |     B
-------------------------------------------------------------------------
    8   |     G     |     F     |     E         |      D        |     C
-------------------------------------------------------------------------
```

8) Explicar el concepto de ejecución superescalar. Relacionarlo con el concepto de Hyper-Threading (Intel SMT).

```
La ejecución escalar consiste en tener múltiples unidades de ejecución,
normalmente en una arquitectura con pipeline. Esto permite que una mayor
cantidad de instrucciones sean ejecutadas al mismo tiempo.

Supongamos un pipeline con tres etapas (Fetch, Decode, Execute), los diagramas
de ejecución serían los siguientes.

Pipelining normal

---------------------------------
 Ciclo  |   F   |   D   |   E   |
---------------------------------
    1   |   A   |       |       |
---------------------------------
    2   |   B   |   A   |       |
---------------------------------
    3   |   C   |   B   |   A   |
---------------------------------
    4   |   D   |   C   |   B   |
---------------------------------
    5   |   E   |   D   |   C   |
---------------------------------
    6   |   F   |   E   |   D   |
---------------------------------


Pipelining + Superescalar

-----------------------------------------------------------
 Ciclo  |   F1  |   F2  |   D1  |   D2  |   E1  |   E2  |
-----------------------------------------------------------
    1   |   A   |   B   |       |       |       |       |
-----------------------------------------------------------
    2   |   C   |   D   |   A   |   B   |       |       |
-----------------------------------------------------------
    3   |   E   |   F   |   C   |   D   |   A   |   B   |
-----------------------------------------------------------

En una microarquitectura superescalar, es posible que terminen existiendo
unidades de ejecución o etapas que se desperdicien, por ejemplo por
dependencias. Una de las formas de solucionar esto es Hyper-Treading (que es
```

```
el nombre comercial de Intel para el Simultaneous Multithreading), que
básicamente consiste en simular la existencia de un segundo procesador, con un
pequeño incremento de hardware (para mantener el estado arquitectural de cada
procesador, o para la lógica de arbitraje necesaria), pero un gran incremento
en el rendimiento, bajo la idea de que las etapas del pipeline van a ser más
utilizado.
```

9) Explicar la idea detrás del concepto de Out-of-Order Execution. - Mencionar ventajas y desventajas de su utilización. - Ejemplificar el diagrama de un flujo de ejecución In-Order vs la misma ejecución Out-of-Order.

```
La idea detrás del concepto de OoOE es que ante dependencias que generen un
pipeline stall, se desperdician etapas, y por lo general hay otras
instrucciones en cola que se podrían estar ejecutando, ya que no tienen
dependencias.

Supongamos que la instrucción B depende del output de A, pero que las
siguientes no. Entonces, si pudieramos tendríamos lo siguiente.

Flujo de ejecución in-order (con stall)
-------------------------------------------------------------------------------
 Ciclo  |   Fetch   |   Decode  |   Fetch Oper  |   Execute     |   Retire
-------------------------------------------------------------------------------
    1   |     A     |           |               |               |
-------------------------------------------------------------------------------
    2   |     B     |     A     |               |               |
-------------------------------------------------------------------------------
    3   |     C     |     B     |     A         |               |
-------------------------------------------------------------------------------
    4   |     D     |     C     |     B         |      A        |
-------------------------------------------------------------------------------
    5   |     D     |     C     |     B         |      -        |     A
-------------------------------------------------------------------------------
    6   |     D     |     C     |     B         |      -        |     -
-------------------------------------------------------------------------------
    7   |     E     |     D     |     C         |      B        |     -
-------------------------------------------------------------------------------
    8   |     F     |     E     |     D         |      C        |     B
-------------------------------------------------------------------------------

Flujo de ejecución out-of-order
-------------------------------------------------------------------------------
 Ciclo  |   Fetch   |   Decode  |   Fetch Oper  |   Execute     |   Retire
-------------------------------------------------------------------------------
    1   |     A     |           |               |               |
-------------------------------------------------------------------------------
    2   |     B     |     A     |               |               |
-------------------------------------------------------------------------------
    3   |     C     |     B     |     A         |               |
-------------------------------------------------------------------------------
    4   |     D     |     C     |     B         |      A        |
-------------------------------------------------------------------------------
    5   |     E     |     D     |     (B) C     |      -        |     A
```

```
-------------------------------------------------------------------------
    6   |     E     |     D     |     B         |      C        |     -
-------------------------------------------------------------------------
    6   |     F     |     E     |     D         |      B        |     C
-------------------------------------------------------------------------
    6   |     G     |     F     |     E         |      D        |     B
-------------------------------------------------------------------------

La mejora puede llegar a ser sustancialmente mejor si se cuenta con ejecución
superescalar.
```

10) ¿Qué son las excepciones imprecisas, y cuales son sus posibles causas?

```
Las excepciones imprecisas surgen al utilizar Out-of-Order Execution. Son
excepciones en donde el estado arquitectural del procesador no es el mismo que
el que tendría si la ejecución hubiera sido in-order. Esto hace que el
programa pueda tomar estados inválidos.

Las causas pueden ser dos, que una instrucción que debería haberse ejecutado
no se haya ejecutado, o que una instrucción que todavía no se debería haber
ejecutado ya se haya ejecutado.

Tomando el ejemplo anterior, se puede interpretar que puede haber una
excepción imprecisa si:

  - C levanta una excepción, pero B todavía no se commiteó, por lo que el
    estado del procesador no es válido.

  - B levanta una excepción, pero C ya se ejecutó, por lo que el estado del
    procesador no es válido.
```

11) ¿Qué es el scoreboarding, y para qué sirve? - Explicar cuáles son las etapas que lo componen, y qué se realiza en cada una de ellas. - Dibujar un diagrama que permita entender cómo se relaciona cada etapa con la siguiente, junto con los demás componentes que fuesen necesarios para implementar el método. - ¿Qué dependencias de datos se ven afectadas por su utilización, en qué etapa y de qué modo? - Mencionar ventajas y desventajas generales.

```
El Scoreboarding es una técnica que permite implementar Out-of-Order
execution, previniendo que la ejecución se vea afectada por las falsas
dependencias (WAW y WAR) surgidas de implementar OoOE. Su función es la de
lograr que cuando hay dependencias de datos (también RAW) o estructurales se
generen stalls, ya sea deteniendo las etapas o unidades funcionales
necesarias, o insertando burbujas. Para ello, divide la etapa de decode en dos
etapas, Issue y Read Operands. La ejecución de una instrucción comprendería
las etapas, todas ellas conectadas a una unidad de scoreboard (marcador,
pizarra), que va guardando la información necesaria de cada etapa (ejemplo,
las utilizaciones de cada operando):

Fetch -> Issue -> Read Operands -> Execute -> Write Results
   |       |          |               |            |
```

```
    ---------------------- Scoreboard ---------------

  1) Fetch: se lee la siguiente instrucción

  2) Issue: se decodifica la instrucción, y se detectan las dependencias
  (lecturas y escrituras), que serán memorizadas para las siguientes etapas.
  Si existe una dependencia WAW, se detiene la ejecución de esa instrucción
  hasta que la escritura sobre el registro de salida se haya realizado.
  También se detiene la ejecución si no hay unidades funcionales disponibles.

  3) Read Operands: se leen los operandos necesarios para ejecutar la
  instrucción. Si existe una dependecia RAW, se detiene la lectura hasta que
  los operandos necesarios se encuentren disponibles, ie. que hayan sido
  escritos por la instrucción correspondiente.

  4) Execute: se ejecutan las instrucciones y se informa cuando se liberan
  unidades funcionales.

  5) Write Results: se escriben los resultados. Si existe una dependencia WAR,
  se detiene la escritura hasta que todas las operaciones que quieran leer de
  esa instrucción lo hayan hecho (que hayan completado su etapa de Read
  Operands).

Las ventaja del scoreboarding es su simpleza y que requiere relativamente poco
hardware adicional en comparación con otros métodos. Las desventajas es que no
soluciona las falsas dependencias (WAR, WAW), sino que sólamente previene los
riesgos asociados, ya que para evitar que el programa tome un estado inválido
las instrucciones deben detenerse hasta que las dependencias se resuelvan, y
esto sólamente sería verdaderamente necesario en las verdaderas dependencias
(RAW). Por ejemplo, las instrucciones con dependencias WAW se detienen en la
etapa de Issue, cuando utilizando alguna otra técnica en realidad podrían
detenerse en la etapa Write Results.
```

12) ¿Qué es el algoritmo de Tomasulo, y para qué sirve? - Explicar cuáles son las etapas que lo componen, y qué se realiza en cada una de ellas. - Dibujar un diagrama que relacione las etapas con el resto de los componentes (ejemplo, Reservation Station, Common Data Bus, Reorder Buffer). - ¿Qué dependencias busca solucionar en comparación con Scoreboarding, y de qué forma lo hace? - Mencionar ventajas y desventajas generales.

```
El algoritmo de Tomasulo es un método de Out-of-Order Execution, cuya
principal idea se basa en el renombrado de registros, algo que permite lidiar
con las falsas dependencias mucho mejor que Scoreboarding.

RAT = Register Alias Table
UF = Unidad Funcional (puede ser punto flotante, escalar, etc)
RS = Reservation Station (cada RS puede albergar una cierta cantidad de
instrucciones pendientes)
CDB = Common Data Bus
SB = Store Buffer
LB = Load Buffer

                    |-> RS -> UF -->|<-------> SB
FETCH----->ISSUE--->|-> RS -> UF -->|
```

```
                    |-> RS -> UF -->|<-------> LB
                    |               |
       RAT <-------->|----->CDB<-----|<-------> RF


En su implementación más básica, sus principales componentes son:

  - Common Data Bus: interconecta todos los componentes. Cada vez que una
    unidad de ejecución completa una instrucción, la informa a través de
    este bus mediante un broadcast, de forma tal que todos los componentes
    necesarios puedan saberlo a la vez.

  - Unidad de Issue (o Instruction Queue): va cargando las instrucciones en
    orden en las Reservation Station, a medida que hayan Reservation Station
    disponibles; es importante notar que cuando existe alguna dependencia, los
    registros se renombran previamente. Esto evita que existan las dependencias
    WAR y WAW.

  - Reservation Station: Son buffers que contienen a las instrucciones. Si
    las instrucciones tienen dependencias sin resolver, son retenidas. A
    medida que las instrucciones se van completando, se van resolviendo
    las dependencias, y entonces se ejecutan. Las dependencias resueltas
    se reciben a través del CDB.

  - Register File: Son los registros del procesador. Cuando una unidad de
    ejecución completa una instrucción cuyo alias de salida se encuentra
    en el RAT, significa que el dato correspondiente debe guardarse en
    el RF.

  - Load/Store Buffer: Son buffers que leen o escriben a memoria. El
    funcionamiento es similar al del RF. Cuando un dato es leído,
    es informada a través del CDB, para que las RS puedan reemplazar los
    tags por el valor correspondiente, y resolver las instrucciones
    dependientes.

  - Register Alias Table: Contiene los alias de los registros que son
    resultados de una instrucción y aún no se han commiteado; ie. los
    registros que son resultado de las instrucciones que se encuentran en
    alguna RS. Esto sirve para saber cómo taggear las dependencias cuando una
    nueva instrucción tiene que ser emitida por la unidad de Issue, y para
    saber dónde guardar los resultados cuando son emitidos por las UF.

Vale mencionar que esta versión básica de Tomasulo permite que existan
excepciones imprecisas. Existe una versión de Tomasulo que utiliza, además, un
Reorder Buffer, que básicamente es un buffer cíclico, que va guardando la
información en-orden de las instrucciones a medida que son sacadas de la
unidad de Issue, y luego se encarga de que las instrucciones sean commiteadas
en orden.
```

13) Explicar qué son los obstáculos de control. - Caracterizar los tipos de discontinuidad que se pueden encontrar en el flujo de una ejecución.

```
Los obstáculos de control son aquellos en donde el flujo de ejecución se ve
```

```
alterado por instrucciones de control. Cuando esto sucede, se dice que el
flujo de ejecución tiene un branch (una ramificación).

Los tipos de saltos se pueden clasificar en:

  Condicional: Son los saltos que dependen del resultado de instrucciones
  previas; en otras palabras, la dirección de destino depende del contexto de
  ejecución del programa.

  Incondicional: Son los saltos que no dependen del resultado de
  instrucciones. La dirección de destino siempre es la misma.

  Directos: La dirección de salto es un parámetro inmediato.

  Indirectos: La dirección de salto está localizada en memoria o en un
  registro, y debe resolverse.

  Return: Son los saltos ocasionados por el retorno de una función; la
  dirección de retorno se encuentra en el stack.

También existen discontinuidades ocasionadas por interrupciones o excepciones,
las cuales no son predecibles.

Cuando un flujo de ejecución tiene un salto condicional, y este termina siendo
realizado, se denomina branch taken.
```

14) ¿Qué sucede con el pipeline cuando el procesador encuentra un salto? Explicar el concepto de Branch Prediction, y relacionarlo con los distintos tipos de branches que pueden encontrarse.

```
En condiciones básicas, cuando se encuentra un salto, hay que descartar todo
el pipeline, ya que todas las instrucciones que se estaban procesando dejan de
parte del flujo de ejecución válido del programa. Esto afecta gravemente el
rendimiento.

Branch Prediction es un conjunto de técnicas destinadas a mitigar el efecto de
los saltos, prediciendo con anticipación la dirección de destino cuando exista
un salto, evitando que haya que descartar el pipeline. Cuando los branches son
incondicionales, predecir el destino es sencillo, ya que es parámetro de la
propia instrucción. Otros tipos de branches (incondicionales, indirectos,
etcétera) no son tan sencillos.

Siempre vale la pena aplicar Branch Prediction, ya que en el peor de los casos
se termina prediciendo una dirección equivocada, y hay que descartar el
pipeline (que es lo mismo que sucede en una situación en la que no se predice
nada).
```

15) Diferenciar los conceptos de predicción estática vs predicción dinámica. Ventajas y desventajas de cada una.

```
La predicción estática son aquellas técnicas de branch prediction que se
```

realizan por fuera del contexto de ejecución. Esto pueden ser optimizaciones durante la compilación del programa, o presunciones sobre los saltos (ejemplo, siempre que haya un salto condicional, el branch va a ser taken). Esto permite resolver los saltos de forma sencilla, y sin contar con hardware ni información adicional. La desventaja es que los niveles de acierto utilizando este tipo de predicciones por lo general no son buenos. Por ejemplo, un salto indirecto no puede ser predicho mediante predicción estática.

La predicción dinámica utiliza información adicional del contexto de ejecución a la hora de resolver las direcciones de destino. Esta información puede ser la dirección de salto de la última vez que se haya corrido esa misma instrucción, si resultó ser branch taken o not taken, el historial de los últimos saltos realizados, etcétera.

16) Explicar las siguientes técnicas. Mencionar ventajas, desventajas y requerimientos de cada una: - Branch Always Not Taken - Branch Always Taken - Backward Taken Forward Not Taken - Profile-Driven Prediction - Delayed Branch Slot - Loop Unrolling - Predicated Execution

Branch Always Not Taken: se asume que siempre que haya un salto condicional, este no será tomado. Por lo general no tiene un buen nivel de acierto, ya que en los loops se realizan varios saltos, y sólamente el último (cuando se viola la condición de la guarda) resulta not taken. No se requiere contar con ningún tipo de soporte especial para realizar este tipo de predicciones.

Branch Always Taken: se asume que siempre que haya un salto condicional, este será tomado. El nivel de acierto es mejor en este caso. No se requiere contar con ningún tipo de soporte especial para realizar este tipo de predicciones.

Backward Taken Forward Not Taken: Es una combinación de los anteriores. Se asume que cuando los saltos son hacia atrás, serán tomados, y cuando son hacia adelante, no serán tomados. Esto es así, porque por lo general los saltos hacia atrás son loops, y los saltos hacia adelante son if, y estadísticamente funciona mejor hacer esta distinción. No se requiere contar con ningún tipo de soporte especial para realizar este tipo de predicciones.

Profile-Driven Prediction: Es una predicción realizada luego de un profiling del programa. El principal problema es que se debe contar con una muestra previa de las ejecuciones del programa, que no siempre es posible, y que además no siempre un subconjunto de ejecuciones van a ser representativos del resto de las ejecuciones. Se requiere (mínimamente) contar con un soporte especial por parte del compilador.

Delayed Branch Slot: ... no lo entendí-
TODO: explicar DBS (Dragon Ball Super?)

Loop Unrolling: Consiste en desenrollar los ciclos, para transformarlos en una secuencia de instrucciones en donde no hayan saltos. Ejemplo, un ciclo que ejecute el código "C" 5 veces, puede ser reemplazado por un código sin ciclos que ejecute "CCCCC". Esto, a pesar de ser una buena técnica, no siempre es posible (la cantidad de repeticiones del ciclo no siempre pueden ser determinados en tiempo de compilación). Además, aumenta el tamaño del programa, lo cual (por ejemplo) puede llegar a afectar el hitrate de la caché.

```
Predicated Execution: consiste en tener instrucciones específicas, definidas
en la ISA, que se ejecuten condicionalmente, para transformar los branches en
un conjunto de instrucciones condicionales. Esto requiere de un soporte
especial por parte del hardware, y también del compilador.
```

17) ¿Qué es el Branch Target Buffer (o Branch Target Predictor), para qué sirve, cómo funciona y cómo está compuesto?

```
El Branch Target Buffer es una memoria que almacena, dada dirección de una
instrucción, la dirección de destino del salto (en caso de que la instrucción
sea un salto). También puede almacenar si el salto fue tomado o no.

Sirve para predecir si una instrucción va a ser de salto y, además, cuál sería
la dirección de destino en ese caso. El funcionamiento es similar al de una
memoria caché.

En la práctica se puede implementar con los últimos N bits de la dirección de
la instrucción (es decir, la dirección modulo N), ya que guardar una tabla de
2^64 direcciones no sería posible, o con una memoria asociativa.

En la práctica el BTB y el BHT (pregunta siguiente) pueden estar combinados.
```

18) ¿Qué es el Branch Prediction Buffer (o Branch History Table, one-level predictor), para qué sirve, cómo funciona y cómo está compuesto? - Explicar cómo funciona un predictor de 1 bit. Explicar su funcionamiento mediante un diagrama de flujo y/o máquina de estados. - Explicar cómo funciona un predictor de 2 bits. Explicar su funcionamiento mediante un diagrama de flujo y/o máquina de estados. - Explicar cómo funcionaría en el caso general un one-level predictor que utilice un contador de n bits. ¿Presentan alguna ventaja respecto a los de menor cantidad de bits?

```
El branch prediction buffer sirve para predecir si un branch será tomado o no.
Por lo general se accede mediante los últimos N bits de la dirección, y
almacena un contador que depende de las ejecuciones de los saltos anteriores.

--

El predictor de 1 bit, almacena 0 si el salto fue not-taken, y 1 si el salto
fue taken. Cada vez que una instrucción se commitea, le informa si la
instrucción fue o no un salto, por lo que se actualiza el valor en caso de ser
necesario.

              -------->taken------>
(not taken)   0                    1 (taken)
              <-----not taken<-----

A la hora de predecir si va a haber un salto, busca la dirección en la tabla.
Si no está, utiliza algún mecanismo de predicción estática (ejemplo Always
Taken, o BTFNT). Si la dirección está, si es un 1 asume taken, y realiza el
salto, mientras que si es un 0 asume not-taken, y no realiza el salto. Luego,
cuando la instrucción se ejecute, en caso de que la predicción haya sido
```

```
equivocada se corregirá el valor en la tabla.

--

El predictor de 2 bits tiene un funcionamiento similar al del predictor de 1
bit, con la diferencia de que el contador utilizado es de dos bits (con
saturación). Es decir, cada vez que una instrucción realiza un salto, se le
suma 1 a esa dirección en la tabla, y cada vez que el salto no es realizado,
se le resta 1, pero teniendo en cuenta que:

  * si está en 00 y se le resta, queda en 00,
  * si está en 11 y se suma, queda en 11,
  * si está en 01 y se le suma, pasa a 11,
  * si está en 10 y se le resta, pasa a 00,


    -------->taken---->
 00                    01
  ^ <----not taken<----  |
  |                      |
not taken              taken
  |                      |
  | ------>taken------>  v
 10                    11
    <----not taken<----

Este predictor ha demostrado estadísticamente tener un mejor ratio de
predicción que el de 1 bit.

--

Un predictor de N bits, en general, es similar al de dos bits, manteniendo las
cuatro condiciones de saturación descriptas anteriormente. Por ejemplo, para
5 bits:

  * si está en 00000 y se le resta, queda en 00000,
  * si está en 11111 y se suma, queda en 11111,
  * si está en 01111 y se le suma, pasa a 11111,
  * si está en 10000 y se le resta, pasa a 00000,
```

19) ¿Qué son los Correlation-Based Branch Predictor? - ¿En qué se diferencian de los one-level predictor de n bits? - Explicar los conceptos de local y global branch correlation.

```
Los predictores de correlación, también llamados predictores de dos niveles,
son aquellos en donde se usa otra información del contexto de ejecución además
de los resultado de "los últimos N saltos".

TODO: Explicar lo que queda de esta parte :(
```

MEMORIA

1) ¿Cuáles son las principales diferencias entre las memorias ROM y las RAM? Dar ejemplos de memorias de cada tipo.

```
Las memorias ROM son memorias que no necesitan electricidad para almacenar los
datos.

Las memorias RAM son memorias que necesitan estar alimentadas permanentemente de
electricidad para almacenar la información.
```

2) Distinguir entre memoria ram estática y dinámica, mencionando ventajas y desventajas de cada una. ¿Cuál de ellas es usada para memoria caché, y cuál es usada para memoria principal? Comparar según: - cómo es el circuito (tamaño, cantidad de componentes electrónicos), - consumo eléctrico, - tiempo de acceso, - capacidad de almacenamiento por chip, - costo de fabricación - uso típico

```
Las memorias RAM dinámicas están hechas por un circuito eléctrico muy simple,
utilizando para cada bit un capacitor para almacenar la carga, y un transistor
para cortar el paso de corriente. La lectura de las mismas es destructiva (ya
que se vacía el capacitor), por lo que requieren de un circuito de
retroalimentación que vuelva a cargar el valor leído luego de la misma.
También requieren un refresco períodico de la información, para evitar que la
carga del capacitor se degrade. El consumo eléctrico es bajo, y la capacidad
de almacenamiento por chip es alta, ya que el circuito es muy simple. El costo
de fabricación también es bajo. Pero el tiempo de acceso es lento, ya que la
lectura no solo implica leer la información, sino retroalimentarla. Su uso
típico es para memoria principal de la PC (lo que normalmente llamamos memoria
RAM).

Las memorias RAM estáticas están compuestas por un flip-flop, que requiere 6
transistores, siendo un circuito más complejo que el de las DRAM. La lectura
no es destructiva, y no necesitan ser refrescados. Tres de los transistores se
encuentran en estado de corte, y tres en estado de saturación, por lo que el
consumo eléctrico es mayor al de las DRAM. Además, al ser la densidad del
circuito mayor, la capacidad de almacenamiento por chip es menor, y el costo
de fabricación es mayor. El tiempo de acceso es mucho mayor que el de las
DRAM, ya que no necesitan de ningún circuito adicional de retroalimentación.
Su uso típico es para memorias caché.
```

3) ¿Cuál es la función de la memoria caché, y cuáles son los dos principios sobre los cuales se fundamenta su utilización?

```
La memoria caché se encuentra entre la memoria principal y el procesador, y su
función es la de proveer de los datos al procesador de forma mucho más rápida,
ya que está hecha de memoria SRAM.
```

```
Su utilización se fundamenta en los principios de localidad espacial y
temporal.

  Principio de localidad espacial: si una dirección de memoria fue accedida
  recientemente, es probable que las direcciones contiguas también sean
  accedidas.

  Principio de localidad temporal: si una dirección de memoria fue accedida
  recientemente, es probable que vuelva a ser accedida nuevamente en el
  futuro.
```

4) Describir los algoritmos de reemplazo más comunes, y relacionarlos con los principios mencionados anteriormente.

```
Dado que la caché tiene un tamaño menor al de la memoria principal (sino no
tendría sentido su uso), se llena rápidamente, lo cual implica que hay que
establecer políticas de desalojo de líneas para hacer lugar a las nuevas
líneas. Las más comunes son:

  Random: Se desaloja alguna línea.

  FIFO (First In First Out): Se desaloja la línea más vieja, según el orden en
  que se las fue alojando

  LRU (Least Recently Used): Se desaloja la línea más vieja, según el orden en
  que van siendo utilizadas.

  LFU (Least Frequenly USed): Se desaloja la línea que tenga menor cantidad de
  usos.
```

5) Diferenciar entre caché de mapeo directo, totalmente asociativa, y asociativa por conjuntos. Dar un ejemplo de cada una, y mencionar ventajas y desventajas.

```
Mapeo directo: cada bloque de memoria se asocia a un único bloque de caché.
Esto se realiza calculando la dirección del bloque de memoria mod k, en donde
k es la cantidad de bloques de caché. La principal desventaja de este esquema
es que en un caso patológico dos (o más) direcciones de memoria podrían estar
compitiendo constantemente por alojar/desalojar el mismo bloque, lo cual
tendría un impacto enorme en el rendimiento.

Totalmente asociativa: cada bloque de memoria no tiene un mapeo fijo, sino que
puede ir a parar a cualquier bloque disponible dentro de la caché. Esto
requiere de una lógica de control específica, e impacta sobre la complejidad,
el tamaño y el precio de la memoria. La principal ventaja es que no suceden
las colisiones que suceden con Mapeo Directo, lo cual evita casos patológicos
y permite implementar políticas de desalojo mucho mejores.

Asociativa por conjuntos: es una mezcla entre los dos tipos de caché
anteriores. Se cuenta con una cantidad S de memorias asociativas, y se divide
la memoria en S sets, de forma tal que cada dirección va a parar a un set
```

```
determinado, y dentro del set se guarda de forma totalmente asociativa.
Mediante este tipo de memoria se logra un buen tradeoff entre las ventajas y
las desventajas de cada tipo.
```

6) Describir las políticas de escritura más comunes, y las ventajas y desventajas de cada una.

```
Write-Through: Cada vez que se escribe una línea, el cambio se escribe también
en memoria principal. Esto tiene como ventaja que mantiene la coherencia entre
la memoria principal y la caché, lo cual puede ser útil en caso de que un
periférico quiera acceder a esa misma memoria, y de que es fácil de
implementar. El inconveniente es que elimina totalmente (para los casos de
escritura) los beneficios que brinda la caché, ya que la latencia termina
siendo la misma (o un poco más, según el caso) que si se trabajara con memoria
principal.

Write-Back: Las líneas no son transmitidas a memoria principal, sino sólamente
hasta que son desalojadas. Esto tiene como ventaja que se preserva el
beneficio de latencia brindado por la caché. Tiene como desventaja que la
lógica para implementarla es más compleja, y que se requieren realizar
controles y operaciones adicionales a la hora de desalojar una línea, o de
utilizar un periférico que acceda a memoria (por ejemplo, quitando esa línea
de la caché, y transformándola en no-cacheable mientras el periférico la esté
utilizando).

Write-Through-Buffered: Es una mezcla entre las políticas anteriores. Se
agrega un buffer de escrituras, y cada vez que se escribe una línea se la
envía a ese buffer. Las escrituras se van realizando de forma autónoma por la
controladora de caché, permitiendo que mientras tanto el procesador siga con
la ejecución de instrucciones. Si el buffer está lleno, se espera hasta que se
haga lugar. La ventaja es que mantiene la coherencia con la memoria principal,
al mismo tiempo que disminuye notablemente el impacto de Write-Through,
teniendo un rendimiento más similar al de memorias Write-Back. La desventaja
es que requiere de todavía más circuito y lógica adicional, por lo que resulta
caro de implementar.
```

7) ¿Qué características generales debe cumplir un sistema para ser considerado SMP? Describir en qué consiste, en un sistema SMP de bus compartido, el problema de la coherencia de caché.

```
Para que un sistema sea Simmetric Multi-Processing, por lo general debe cumplir con:

  Existen dos o más procesadores de capacidades similares.

  Todos los procesadores pueden ejecutar las mismas funciones (esto se
  entendería como, tienen la misma ISA).

  Existe un sistema operativo que posibilita la interacción entre los
  procesadores.

  Los procesadores comparten la memoria principal, y los dispositivos de
  entrada y salida, mediante un mismo bus que los conecta, de forma tal que el
```

```
  tiempo de acceso a memoria es el mismo para todos ellos.

Por lo general el bus compartido funciona de árbitro, secuenciando las
operaciones de los distintos procesadores, y haciendo de cuello de botella.
Por este motivo, el sistema SMP no escala a una gran cantidad de procesadores
(ejemplo, 1024 procesadores).

En este contexto, el problema de la coherencia de caché se da cuando dos
procesadores tienen en sus respectivas caché distinta información sobre una
misma línea de memoria. Un ejemplo básico de esto es, P1 lee la dirección X,
P2 lee X, y luego P1 escribe X. Incluso en un esquema de escritura Write-
Through, a priori (sin establecer un mecanismo de coherencia) nada asegura que
P2 y P1 vayan a tener el mismo valor en la dirección X.
```

8) ¿Qué condiciones debe cumplir un sistema de memoria para ser considerado coherente? Explicar las propiedades de coherencia y consistencia, y relacionarlas con los conceptos de propagación y serialización.

```
Para que un sistema de memoria sea coherente debe cumplir con las características
de:

  Coherencia: Todos los procesadores comparten la misma información. Esto está
  relacionado con la correcta propagación de la información, a modo de
  ejemplo, "cuando un procesador realiza una escritura, debe propagarla a las
  otras cachés". Una implementación muy básica de propagación (ejemplo, avisar
  sobre todas las operaciones a todos los procesadores) puede degradar
  enormemente el rendimiento.

  Consistencia: Las operaciones realizadas sobre una determinada línea de
  memoria son vistas por todos los procesadores en el mismo orden relativo,
  que además debe ser coherente con el orden respecto a las operaciones sobre
  las otras líneas de memoria. Es decir, si sucede en el siguiente orden que:
  P1 escribe en X, P2 escribe en X, P1 lee de X, P3 escribe en X y P1 lee de
  X, cada procesador debe comportarse respetando este orden. Dado que los
  procesadores trabajan de forma concurrente, garantizar esto no es sencillo.
  Ejemplo muy básico: si el protocolo es "escribir y luego avisar al resto", y
  P1 y P2 escriben al mismo tiempo en la dirección X, lo que terminaría
  sucediendo es que P1 se quedaría con el valor escrito por P2, y P2 se
  quedaría con el valor escrito por P1, ya que cada uno tendría las lecturas
  en el orden inverso.
```

9) ¿Qué soluciones existen para los problemas de coherencia de caché en sistemas monoprocesador? Ejemplo, cuando un dispositivo DMA escribe a memoria. - ¿Qué ventajas y desventajas traen estas soluciones? - ¿Requieren algún tipo de soporte de hardware o software especial? - ¿Por qué no resulta práctico implementar esto mismo en sistemas SMP?

```
Las soluciones posibles son varias:
```

Desalojar las líneas de caché antes de realizar DMA. Esto se hace manualmente, invalidándolas por software. Aumenta la complejidad del código. A priori, no es posible prever cuando otro procesador va a querer usar una determinada línea de memoria, por lo que sin un protocolo de comunicación destinado a mantener la coherencia de caché, esto no sería posible.

Que el DMA pueda realizarse directo sobre la caché. Esto requiere contar con hardware especial. Aumenta la complejidad del hardware. Si cada procesador accede a la información directo en la caché del procesador 0 (posible implementación), entonces habría que eliminar la caché de los otros procesadores, y no tendría sentido usar caché. Además, esto afectaría el rendimiento del procesador 0, ya que estaría recibiendo constantes pedidos de lectura de otros procesadores.

Definir instrucciones que permitan setear a una línea de memoria como no-cacheable. Requiere una arquitectura especial, y aumenta la complejidad del código. Para implementar esto en SMP habría que definir todas las direcciones de memoria en uso como no cacheables, es decir, no usar caché.

Definir arquitecturas especiales en donde la DMA se pueda realizar sobre ciertos sectores de memoria que estén definidos como no-cacheables. Requiere contar con una arquitectura y posiblemente un hardware especial. Los accesos a estas direcciones de memoria serían lentos. Para implementar esto en SMP habría que definir todas las direcciones posibles como no cacheables, o sea, no usar caché.

10) Describir la idea detrás de los protocolos de directorio y los protocolos de snooping, comparando ventajas y desventajas de cada uno de ellos.

Los protocolos de directorio se basan en que la coherencia de caché esté regida por un dispositivo externo (directorio). Son populares en sistemas grandes con muchos procesadores. Cada procesador se comunica con el directorio, quien es el que toma las decisiones de coherencia; cada bloque de Caché tiene un único Owner, que está coordinado por este Directorio. Requieren de hardware adicional para su implementación, y desde el vamos presentan una latencia mayor, ya que las operaciones deben ser coordinadas por un único directorio. Se utiliza mucho en entornos de clusters (redes de computadoras de muchos procesadores).

Los protocolos de snooping se basan en un bus único, que hace que las transmisiones sean secuenciales (y por lo tanto los requests). Cada procesador se encarga de espiar el bus, y de hacer broadcast de las operaciones necesarias para permitir la coherencia, por lo que no hay un único dispositivo coordinando todas las operaciones. No requiere hardware adicional, aunque al no haber un dispositivo que coordine requiere un soporte explícito por parte de los procesadores, y los protocolos utilizados pueden llegar a ser más complejos. Se utiliza en entornos en donde hay pocos procesadores (ejemplo, 4), pero no escala bien, ya que al utilizar todos el mismo bus, este actúa como cuello de botella.

11) Describir las diferencias entre los protocolos de invalidación en escritura, vs los de actualización

en escritura. Ejemplificar, ¿qué ocurre en cada caso si un procesador desea leer de su caché una posición de memoria que fue previamente modificada por otro procesador?

```
Los protocolos de invalidación en escritura emiten mensajes de broadcast
invalidando el bloque correspondiente en las otras cachés cuando hay una
escritura en una caché. Así, si una caché quiere leer un bloque que acaba de
ser modificado por otra caché, tienen que leerlo nuevamente.

Los protocolos de actualización en escritura, emiten mensajes de broadcast con
los nuevos datos para que las otras cachés los actualicen, evitando que las
otras cachés tengan que leerlos nuevamente (ya que simplemente los actualizan,
y siguen marcados como válidos). Si una caché quiere leer un bloque, lo puede
hacer, siempre que lo tenga en estado válido.

Los protocolos de actualización en escritura pueden tener un rendimiento mucho
menor a los de invalidación ya que, por ejemplo, múltiples escrituras
consecutivas a una misma dirección de un bloque de la caché de un procesador,
hace que se emitan múltiples mensajes broadcast de update (a diferencia de un
protocolo de invalidación, en donde sólamente se emitiría un mensaje). Peor
aún, múltiples escrituras en distintas direcciones del mismo bloque (que es un
caso mucho más común, por ejemplo, imaginemos un arreglo siendo copiado),
también emitirían múltiples mensajes de broadcast.
```

12) Explicar cómo funciona un protocolo de snooping, de dos estados (válido, inválido). - ¿Cumple con las condiciones de coherencia y consistencia? - ¿Qué políticas de escritura soporta? - ¿Qué ventajas y desventajas trae usar este protocolo?

```
Un protocolo de snooping de dos estados funciona de forma similar a un
protocolo de coherencia para un solo procesador con write-through. Cada bloque
de la caché se marca con un estado:

  Válido: indica que los valores de los datos contenidos en ese bloque son
  coherentes con la memoria principal, y con los otros cachés. Esto significa
  que la caché puede leer esta información, y para escribirla deberá informar
  previamente a las otras cachés que deben invalidar el dato. Esto será
  necesario para cada escritura que se realice.

  Inválido: indica que los valores de ese bloque son incoherentes, es decir,
  para su utilización se necesitará leerlos previamente.

Cumple coherencia y consistencia, ya que fuerza a que cada vez que un
procesador desee hacer una escritura, tenga que invalidar previamente los
bloques en las otras cachés. Y la serialización viene dada por el bus único.

Sólamente soporta políticas de escritura write-through.

La ventaja de este protocolo es que no requiere contar con estados
adicionales, y la lógica es sencilla, similar a la de un protocolo de
coherencia monoprocesador. La desventaja es que sólamente funciona con write-
through, por lo que de por sí arrastra todas las desventajas de write-through.
Al haber un bus único, y estar serializadas las operaciones de escritura, esto
genera un cuello de botella incluso en situaciones en donde múltiples
```

```
procesadores se encuentren escribiendo al mismo tiempo, pero cada uno en
distintos bloques de memoria.
```

13) Explicar en qué consiste y cómo funciona el protocolo MSI, detallando cada estado. - Explicar la diferencia con el protocolo de snooping de dos estados. - ¿Cumple con las condiciones de coherencia y consistencia? - ¿Qué políticas de escritura soporta? - ¿Qué ventajas y desventajas trae usar este protocolo?

```
El protocolo MSI se puede considerar una extensión del protocolo de snooping
de dos estados, separando el estado Válido en los estados Modificado o
Compartido (Shared).

Se describen los estados, desde el punto de vista de la caché que los
contiene:

  Modificado: los valores de este bloque han sido modificados por este
  procesador. Esto significa, además, que todos los otros cachés tienen el
  bloque marcado como inválido. La caché puede hacer escrituras o lecturas
  libremente sobre el bloque.Si otra caché realiza una lectura de este bloque,
  el estado pasará a Compartido. Si otra caché realiza una lectura exclusiva
  de este bloque (o sea, una lectura con intenciones de escribir), el estado
  pasará a Inválido. Cuando este bloque sea desalojado (o cambie a otro
  estado), siempre se deberá hacer previamente un write-back de la
  información a memoria principal.

  Compartido: los valores de este bloque no han sido modificados, y además
  puede estar compartido en una o más cachés de otros procesadores (o en
  ninguna). Se pueden hacer lecturas libremente. Cada vez que se desee hacer
  una escritura, se deberá emitir un pedido de lectura exlusiva (Read For
  Ownership), para poder pasar al estado modificado, invalidando a todas las
  otras cachés. Asimismo, si otra caché realiza una lectura exclusiva, se
  pasará al estado inválido.

  Inválido: los valores de este bloque no son válidos. Si se desea utilizar
  este bloque, se deberá emitir un pedido de lectura, o de lectura exclusiva,
  según el caso.

Este protocolo funciona tanto para políticas write-through como write-back.

No tengo ganas de explicar por qué cumple con las condiciones de
coherencia/consistencia, pero la explicación informal es similar a las
anteriores: la serialización de las operaciones viene dada por el bus único, y
la propagación por el protocolo. (TODO: ?)

Las ventajas sobre el protocolo de dos estados son que al haber un estado
modificado, una caché puede hacer múltiples escrituras sobre un bloque, sin
necesidad de emitir múltiples broadcast, sino sólamente una invalidación la
primera vez. Una desventaja es que cada vez que quiera pasar del estado
compartido a modificado, deberá emitirse un Read For Ownership, aún cuando es
posible que en realidad ninguna otra caché tenga ese bloque de memoria (por lo
cual se estaría emitiendo un broadcast innecesario). Es decir, no existe la
noción de que un determinado bloque exista sólamente en una de las cachés.
```

TODO:

14) Explicar cómo funciona el protocolo MESI, detallando cada estado. Hacer un diagrama. - Explicar la diferencia diferencia con MSI, ¿qué ventajas presenta? - ¿Cumple con las condiciones de coherencia y consistencia? - ¿Qué políticas de escritura soporta? - Dibujar un diagrama de estados. - ¿Qué desventajas presenta este protocolo? ¿Existen soluciones?

```
El protocolo MESI se puede considerar una extensión del protocolo MSI, el cual
incorpora el estado Exclusivo. Las transiciones entre estados son similares.

El estado Exclusivo indica que el bloque se encuentra en esta caché, y
sólamente en esta, por lo que es posible hacer transición hacia el estado
Modificado sin necesidad de emitir un Read For Ownership.
```

CASOS PRÁCTICOS / PAPERS

1) Explicar la microarquitectura P6 (Three Cores Engine).

2) Explicar la arquitectura Netburst, y las ventajas frente a P6.

3) Explicar las ventajas de HyperThreading.