PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2343 — Arquitectura de Computadores — 1/2018
I3

Martes 5 de Junio de 2018

La interrogación se compone de 3 preguntas. Debe responder cada una en una hoja separada, aunque puede utilizar más de una hoja para responder a una pregunta. Si no responde a una pregunta, debe entregar una hoja en blanco para ella. Recuerde poner su nombre en todas las hojas de respuesta.

Al momento de entregar, **evite doblar bordes o corchetear las hojas**, si no que sencillamente indique cuántas hojas son para esa pregunta y qué hoja es cada una en caso de utilizar más de una y procure que su hoja esté lo más lisa posible.

1. **Paralelismo simple**.

   1.1) Determine el número de ciclos que se demora el siguiente código, detallando en un diagrama los estados del *pipeline* por instrucción. El *pipeline* tiene *forwarding* entre todas sus etapas, el manejo de *stalling* es por software (instrucción `NOP`) y predicción de salto asumiendo que **siempre** ocurre. Indique en el diagrama cuando ocurre *forwarding*, *stalling* y *flushing*. **[3 pts]**

```
DATA:
    n     3
    index 2
    prev1 6
    prev2 1
    res 0
CODE:
    main:
        MOV A,(n)
        MOV B,(index)
        JEQ end
        INC B
        MOV (index),B
        JMP main
    end:
        MOV A,(prev1)
        MOV B,(prev2)
        ADD A,B
        MOV (res),A
```

Respuesta:
Para esta pregunta se aceptan dos posibles respuestas, dado el hecho de que **no se especifica** si al tener predicción de salto asumiendo que siempre ocurre se carga la línea correspondiente en el registro `PC` de inmediato o no. Entonces, se aceptan dos casos:

- Se asume que **no** se carga la línea de inmediato, por lo que es necesario esperar por tres ciclos haciendo uso de *stalling*.
- Se asume que *sí* se carga la línea de inmediato, por lo que posterior al salto sigue la primera instrucción del *label* correspondiente.

A partir de esto, vemos los dos posibles *pipeline*:

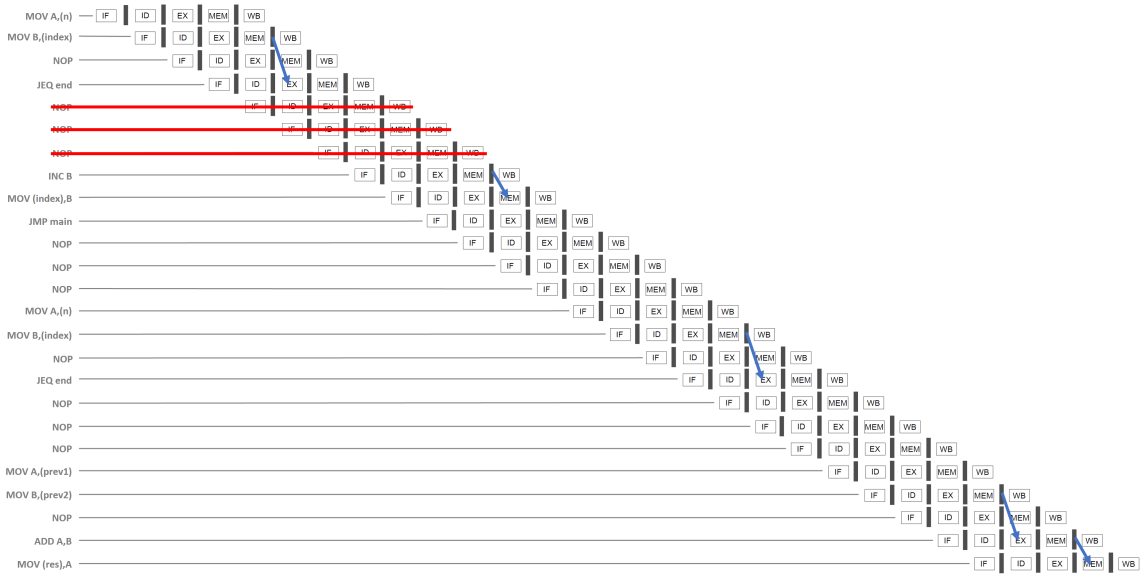

Figura 1: *Pipeline* del primer caso, con 29 ciclos.

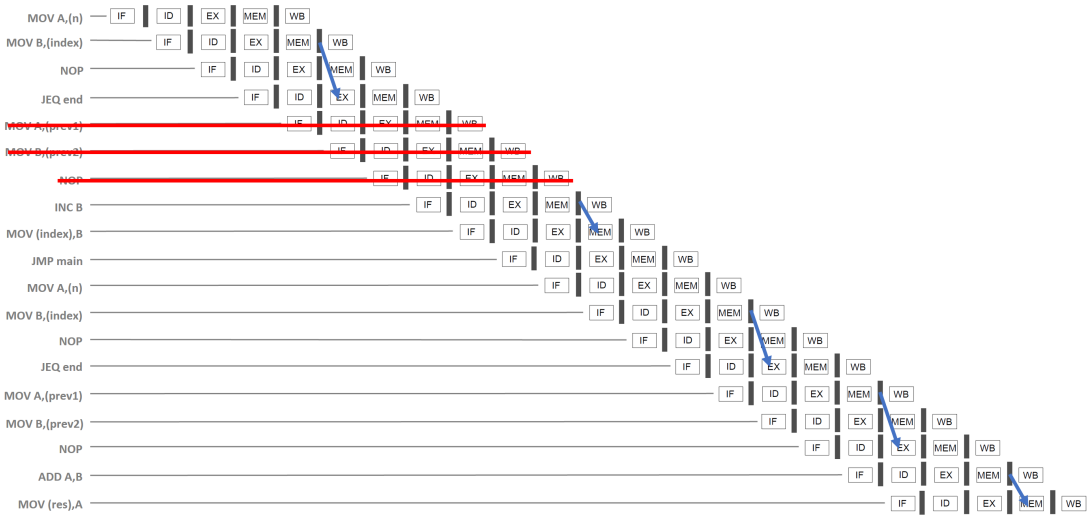

Figura 2: *Pipeline* del segundo caso, con 23 ciclos.

En base a estos, la distribución de puntaje es la siguiente:

- 0.5 pts por indicar correctamente el número de ciclos. Descuento de 0.1 pts de esta cantidad por una diferencia de una unidad en la cantidad, dependiendo del caso escogido, *i.e.*:

$$desc = \max\{0.1 \times |C_{\text{real}} - C_{\text{respuesta}}|, 0.5\}$$

- 0.5 pts por identificar el *flush* correctamente.
- 1/3 pts por cada *stalling* identificado correctamente (sin contar los utilizados en el primer caso para esperar la instrucción siguiente).
- 0.2 pts por cada *forwarding* identificado correctamente.

1.2) Al hacer un computador con *pipeline*, por ejemplo en el caso del computador básico, ¿qué componentes se agregan? **[1 pt]**
Respuesta:
Registros de separación entre cada fase (1/3 pts), unidades de control de *hazard/forwarding* (1/3 pts), unidad de salto (1/3 pts).

1.3) ¿Por qué se permite hacer *forwarding* en algunos casos en un *pipeline* en lugar de hacer *stalling* de la `CPU`? **[1 pt]**
Respuesta:
Debido a que es posible determinar que esos casos requieren el dato de forma rápida y es mejor permitir la ejecución en lugar de parar la `CPU` por uno o más ciclos.

1.4) ¿Cuál sería un riesgo a considerar al implementar un *pipeline* de gran profundidad (*i.e.* muchas etapas)? Justifique en base al rendimiento esperado. **[1 pt]**
Respuesta:
El riesgo que puede surgir es la implementación de la componente de predicción de saltos (0.5 pts). Si esta se ubica en una etapa cercana al final, su predicción errónea implicaría una pérdida considerable de ciclos y perjudicando la eficiencia de la ejecución del programa paralelizado (0.5 pts).

2. **Coherencia de *caché***

2.1) ¿Por qué podría ser más conveniente utilizar una arquitectura `NUMA` por sobre una `UMA` de bus compartido al tener un esquema de memoria compartida con una gran cantidad de procesadores? **[1 pt]**
Respuesta:
Porque en un esquema `UMA` los mensajes de los controladores por procesador podrían saturar el bus compartido, haciendo más viable una arquitectura `NUMA` para no caer en ese problema.

2.2) Mencione una ventaja y una desventaja de actualizar las líneas de *caché* desactualizadas en vez de invalidarlas. **[1 pt]**
Respuesta:
Ventaja: Permite que el computador aproveche al máximo la ventaja de tener datos en la *caché*.
Desventaja: Es muchísimo más complejo implementar un protocolo de coherencia que cubra este caso en lugar de simplemente invalidar.

2.3) El protocolo `MESIF` corresponde a una variante del protocolo `MESI` estudiado en clases. Este posee el mismo comportamiento, salvo por la existencia de un nuevo estado `F`. Este es utilizado para indicar que uno y solo uno de los controladores de *caché* será el encargado de compartir una línea compartida que posea el mismo valor que en memoria (es decir, si alguno de los procesadores con una copia del dato lo modifica, ya no se puede compartir). Esto permite que, en caso de que solo se hagan lecturas de un recurso compartido, la transferencia de datos se haga entre *cachés* y no desde la memoria principal. En base a este protocolo, conteste las siguientes preguntas:

a) ¿Por qué esto es más eficiente que el protocolo `MESI`? Justifique. **[1 pt]**
Respuesta:
Porque la transferencia realizada entre *cachés* es más rápida que la transferencia desde y hacia la memoria principal, considerando las características de velocidad y tamaño de una *caché*.

b) ¿Cómo se podría asegurar que sea una sola *caché* la que posea el estado `F` para un recurso compartido? Comente considerando el procedimiento del protocolo `MESI` para recursos compartidos (en particular, para los estados `E` y `S`). **[3 pts]**
Respuesta:
Cuando un recurso se comparte por primera vez se asigna el estado `E` para indicar que es exclusivo. Luego, cuando otro procesador solicita su lectura, tanto para dicho procesador como para el primero el estado se convierte en `S`, pues está compartido. En el protocolo `MESIF` se puede establecer, en cambio, que el primer procesador cambie a estado `F` en vez de `S` para indicar que será el encargado predeterminado para compartir el recurso entre las *cachés*, siempre que no existan solicitudes de escritura que hagan obsoleta su copia. Naturalmente, el resto de los procesadores que soliciten la lectura del recurso para almacenarlo en sus *cachés* mantendrán un estado `S`, efectuando la transferencia desde la *caché* con la línea en estado `F`. De esta forma se logra que solo una *caché* posea estado `F` para una línea en particular (en este caso, la primera en acceder a ella).

Importante: No es necesario que este procedimiento sea exactamente igual en la respuesta, se da el puntaje según la siguiente distribución:

- 2 pts. si la propuesta asegura que **al menos** una *caché* posea el estado `F`, efectuando transferencias entre *cachés*.
- 1 pt. si la propuesta asegura que **solo una** *caché* posea este estado.

3. **Sección crítica**

3.1) Considera el siguiente algoritmo para el problema de la sección crítica (SC):

wantP = 0, wantQ = 0	
P	Q

while true:
sección no crítica
<if wantQ == -1:
wantP = -1
else:
wantP = 1>
<await wantQ $\neq$ wantP>
SC
wantP = 0
	while true:
	sección no crítica
	<if wantP == -1:
	wantQ = 1
	else:
	wantQ = -1>
	<await wantP $\neq$ -wantQ>
	SC
	wantQ = 0

Cuadro 1: Código de los procesos P, Q.

a) Nota que las sentencias **if** y **await** son indivisibles, o atómicas (están encerradas entre $<$ y $>$). Muestra que la propiedad de exclusión mutua se cumple. **[2 pts]**
Respuesta:
Para que ambos procesos hayan entrado a la sección crítica, se debe cumplir la condición de ambos **await** simultáneamente, lo cual no es posible: al salir del **if** tanto *wantP* como *wantQ* serán 1 o -1; en consecuencia, una de estas variables debe ser ambos valores para cumplir con $(Q \neq P) \wedge (P \neq \neg Q)$.

b) Si en particular la sentencia **if** no fuese atómica, muestra que la propieded de exclusión mutua no se cumpliría. **[2 pts]**

3.2) De las cuatro propiedades importantes estudiadas en el problema de la sección crítica, ¿cuál se podría relajar y aún así asegurar una ejecución correcta del fragmento de código protegido? Justifique su respuesta. **[2 pts]**
Respuesta:
La demora innecesaria o *busy waiting* (1 pt). Se perdería mucha eficiencia al estar gastando tiempo en esperar, pero se aseguraría una entrada a la sección crítica de un proceso a la vez, sin producir bloqueos (1 pt). Se pueden aceptar otras respuestas (con excepción de exclusión mutua) siempre que estén correctamente justificadas.