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

IIC2343 - Arquitectura de Computadores (I/2026)

Interrogación 3

Pauta de evaluación

Pregunta 1: Jerarquía de memoria (6 ptos.)

(a) (1 pto.) En la función de correspondencia *fully-associative*, ¿qué representan los bits de *tag*? ¿Por qué no se puede usar una cantidad menor de bits para este valor?

> **Solución:** En este caso, el *tag* corresponde al **índice del bloque al que pertenece la dirección de memoria**. Se necesitan todos estos bits porque, como en cada línea de la caché se puede almacenar cualquiera de los bloques de la memoria principal, se necesita el índice completo para distinguir un bloque de otro. Si se usaran menos, dos bloques podrían tener coincidencia de *tag* a pesar de ser distintos, generando *hits* incorrectamente. Se otorgan **0.5 ptos.** por señalar la representación del *tag* y **0.5 ptos.** por justificar el uso completo de la secuencia de bits.

(b) (1 pto.) ¿Qué principio de localidad es explotado a través de la política de reemplazo LRU? Justifique su respuesta.

> **Solución:** Se explota el **principio de localidad temporal**. Este señala que un dato obtenido de memoria volverá a ser utilizado en el corto plazo. Como la política de reemplazo LRU se asegura de que se mantengan en caché los bloques de memoria accedidos más recientemente, esto aprovecha el principio de localidad antes mencionado. Se otorgan **0.5 ptos.** por indicar el principio de localidad correcto, y **0.5 ptos.** por justificación.

(c) (4 ptos.) Suponga que, para una memoria principal de 2048 palabras de 1 byte, posee una caché de 16 líneas y 16 palabras por línea con el siguiente estado intermedio:

Línea	*Valid*	*Tag*	*Time*
0	1	4	8
1	1	7	9
2	1	6	14
3	0	5	-
4	1	2	7
5	1	15	6
6	1	6	11
7	1	4	10

Línea	*Valid*	*Tag*	*Time*
8	1	15	12
9	1	8	13
10	1	12	5
11	1	4	4
12	1	4	2
13	1	11	3
14	1	3	1
15	0	1	-

La columna "Línea" representa el índice de línea; "*Valid*" representa el *valid bit*; "*Tag*" representa el valor decimal del *tag* que indica el bloque de memoria almacenado en la línea; y "*Time*" representa el tiempo **desde el último acceso a la línea**. A partir de este estado, se realiza el acceso a memoria de las siguientes direcciones: 0x29A, 0x0FB, 0x33C, 0x7BD, 0x7CE. Indique, para cada acceso, si existe un *hit* o *miss* y, si corresponde, la línea de la caché que es modificada para la función de correspondencia *2-way associative*. Asuma una política de reemplazo LRU. Para responder, complete las tablas adjuntas al enunciado. El *tag* de cada línea lo puede escribir en base binaria, decimal o hexadecimal. No necesita indicar el *timestamp* final por línea, pero sí debe considerar su valor en caso de reemplazos.

Solución: Al tener una memoria de 2048 palabras, se necesitan $\log_2(2048) = 11$ bits para cada dirección. Luego, al ser las líneas de 16 palabras, se requieren $\log_2(16) = 4$ bits de *offset* para ubicar una palabra dentro de una línea. Por otra parte, en una caché *2-way associative* se tienen conjuntos de 2 líneas, lo que deriva en un total de $\frac{16}{2} = 8$ conjuntos y, de esta forma, $\log_2(8) = 3$ bits de índice de conjunto. Esto, finalmente, resulta en $11 - 4 - 3 = 4$ bits de *tag*. Con esto en consideración, se obtiene el siguiente resultado de la secuencia de accesos a memoria:

1. Acceso a dirección 0x29A = 666 = 0101|001|1010b. Resulta en *miss* al no existir línea válida con *tag* 0101b = 5 en el conjunto 001 (líneas 2-3). Se copia el bloque sobre la línea 3 al estar desocupada. La línea 2 sigue siendo la que posee mayor *timestamp* dentro del conjunto, en caso de requerir un reemplazo.

2. Acceso a dirección 0x0FB = 251 = 0001|111|1011b. Resulta en *miss* al no existir línea válida con *tag* 0001b = 1 en el conjunto 111 (líneas 14-15). Se copia el bloque sobre la línea 15 al estar desocupada. La línea 14 sigue siendo la que posee mayor *timestamp* dentro del conjunto, en caso de requerir un reemplazo.

3. Acceso a dirección 0x33C = 828 = 0110|011|1100b. Resulta en *hit*, ya que la línea 6 posee *tag* igual a 0110b = 6 en el conjunto 011 (líneas 6-7). La línea 7 se convierte en la que posee mayor *timestamp* dentro del conjunto en caso de que se requiera un reemplazo.

4. Acceso a dirección 0x7BD = 1981 = 1111|011|1101. Resulta en *miss* al no existir línea válida con *tag* 1111b = 15 en el conjunto 011 (líneas 6-7). Se copia el bloque sobre la línea 7, al ser esta la de mayor *timestamp*. Ahora, la línea 6 es la que posee mayor *timestamp* dentro del conjunto en caso de requerir un reemplazo.

5. Acceso a dirección 0x7CE = 1998 = 1111|100|1110b. Resulta en *hit*, ya que la línea 8 posee *tag* igual a 1111b = 15 en el conjunto 100 (líneas 8-9). La línea 9 sigue siendo la que posee mayor *timestamp* dentro del conjunto en caso de requerir un reemplazo.

A continuación, se muestra el estado final de la caché:

Línea	*Valid*	*Tag*
0	1	4
1	1	7
2	1	6
3	1	5
4	1	2
5	1	15
6	1	6
7	1	15

Línea	*Valid*	*Tag*
8	1	15
9	1	8
10	1	12
11	1	4
12	1	4
13	1	11
14	1	3
15	1	1

Se otorgan **0.8 ptos.** por acceso correctamente descrito (*i.e.* obtención de *tag*; determinación de *hit* o *miss* y selección de línea a reemplazar). Si se reemplaza la línea incorrecta por *timestamp*, se otorgan **0.4 ptos.** en el acceso correspondiente.

Excepcionalmente, se otorgan **3 ptos.** del total si se realiza **todo** el procedimiento de forma correcta, pero con error de arrastre en el cálculo de bits de *offset*, índice de conjunto o *tag*; o bien si se parte contando del conjunto 1 y no del conjunto 0 al momento de seleccionar las líneas.

Pregunta 2: Coherencia de Caché (6 ptos.)

Suponga que posee una arquitectura multiprocesador de memoria compartida con 3 CPU que poseen su propia caché y que siguen el protocolo MESI para asegurar consistencia. La memoria contiene los siguientes datos de variables; mientras que CPU0, CPU1 y CPU2 ejecutan los siguientes programas:

Dirección	*Label*	Valor
0x00	var1	127
0x01	var2	126
...	...	...
0xFE		0
0xFF		0

```
// CPU0              // CPU1              // CPU2
POP A                MOV A, (var1)     MOV A, 1
                     ADD A, 1          MOV B, 2
POP B                MOV B, (var2)     PUSH A
                     ADD B, A          PUSH B
ADD A, B             MOV (B), A        POP B
MOV (var1), A        MOV (var2), B
```

Asumiendo que cada dirección se almacena en una línea distinta, y que el contador `SP` posee un valor inicial igual a 255 en **todas** las CPU, indique el estado de cada línea de las caché (`M`/`E`/`S`/`I`) para cada ciclo completando la tabla adjunta, asumiendo que todas las líneas parten en estado `I` (ciclo 0):

	CPU0				**CPU1**				**CPU2**			
Ciclo	**0x00**	**0x01**	**0xFE**	**0xFF**	**0x00**	**0x01**	**0xFE**	**0xFF**	**0x00**	**0x01**	**0xFE**	**0xFF**
0	I	I	I	I	I	I	I	I	I	I	I	I
1												
2												
3												
4												
5												
6												

IMPORTANTE: Puede usar esta tabla de borrador, pero debe entregar la que se encuentra adjunta en la hoja de respuestas.

Solución: A continuación se muestra la tabla final esperada por cada iteración:

	CPU0				**CPU1**				**CPU2**			
Ciclo	**0x00**	**0xFD**	**0xFE**	**0xFF**	**0x00**	**0xFD**	**0xFE**	**0xFF**	**0x00**	**0xFD**	**0xFE**	**0xFF**
0	I	I	I	I	I	I	I	I	I	I	I	I
1	I	I	I	I	E	I	I	I	I	I	I	I
2	S	I	I	I	S	I	I	I	I	I	I	I
3	S	I	I	I	S	E	I	I	I	I	I	M
4	S	S	I	I	S	S	I	I	I	I	M	M
5	S	S	I	I	S	S	M	I	I	I	I	M
6	M	I	I	I	I	M	S	I	I	I	S	M

Para llegar al estado final correcto, es importante haber tomado en cuenta lo siguiente:

- Al ser `SP` parte de una CPU, su valor es **independiente** de lo que ejecuten otras. Es decir, que una CPU realice un `PUSH` o un `POP` no afecta el valor de `SP` de otra.
- La ejecución de `POP A` y `POP B` toma dos ciclos. La lectura del valor de memoria se realiza en el **segundo ciclo**, dado que el primero lo único que hace es incrementar el valor de `SP`.
- Al hacer `POP` sin haber hecho un `PUSH` antes, la operación generará un *overflow* sobre el contador `SP`, causando que su valor cambie de 255 a 0. Esto implica que, en la CPU0, al ejecutar `POP` realizará una lectura de memoria sobre la dirección 0 en el segundo ciclo y luego sobre la dirección 1 en el cuarto ciclo, *i.e.* lee las variables `var1` y `var2`.
- Si una CPU hace una lectura sobre una línea ya modificada por ella misma, esta sigue estando en dicho estado y solo cambia en dos escenarios: (1) si otra CPU realiza una lectura sobre ella, cambiando a estado *Shared*; o (2) si otra CPU realiza una escritura sobre ella, cambiando a estado *Invalid*.

Se otorga **1 pto.** por cada iteración (fila) con estado final correcto para todas las CPU. Se otorgan **0.7 ptos.** por iteración (fila) si se tiene como máximo una celda con estado erróneo; y **0.3 ptos.** si se tienen como máximo dos celdas con estado erróneo. En otro caso, no se otorga puntaje en la iteración.

Para efectos de corrección, toda celda vacía se considerará en estado inválido, a excepción del caso en el que no se haya contestado la pregunta.

Si una celda se modifica a un estado erróneo en una iteración y se mantiene en la siguiente, solo se descuenta en la fila donde ocurrió el error. En cambio, si debía cambiar su estado en la siguiente iteración y se mantiene en el estado erróneo, se descuenta en ambas filas.

Por último, si un error de interpretación de código genera un estado erróneo en múltiples celdas, se aplica el descuento solo sobre una de ellas si los cambios de todas las celdas son coherentes con la interpretación.

Pregunta 3: Paralelismo a nivel de instrucción (ILP) (6 ptos.)

(a) (2 ptos.) Suponga que decide modificar el computador básico con *pipeline* de la siguiente forma: elimina el registro intermedio `EX/MEM` y fusiona las etapas `Execute` y `Memory` en una sola: "`Execute-Memory`", por lo que ahora el registro intermedio `MEM/WB` pasa a llamarse `EX-MEM/WB`. Describa dos implicancias de esta modificación en términos de ciclos de ejecución, frecuencia de *clock* y/o *hazards*.

Solución: Esta modificación tiene varias implicancias. A continuación, se listan algunas de ellas:

- Disminuye la cantidad de ciclos de ejecución por instrucción y, así, la cantidad de ciclos de ejecución de un programa. No obstante, si se considera que la etapa `Memory` suele ser la de mayor tiempo de ejecución y a partir de la cual se establece la frecuencia del *clock*, esta modificación probablemente también implica un **aumento en el periodo** (*i.e.* un aumento en la duración de un ciclo) para que esta nueva etapa alcance a estabilizarse.
- El *flushing* en una predicción de salto errónea ahora implicaría la pérdida de dos ciclos de ejecución y no tres, dado que ahora en la tercera etapa de ejecución se determinaría el valor de la señal $\texttt{L}_{\texttt{PC}}$.
- Las *forwarding units* 1 y 2, en caso de detectar *hazards* de datos, ahora hacen el *forwarding* directamente con las señales almacenadas en el registro intermedio `EX-MEM/WB`.
- Se elimina el caso de *hazard* de datos en el que hay una dependencia entre la primera y tercera instrucción de un programa: cuando la tercera instrucción se encuentra en `Instruction Decode`, la primera se encuentra finalizando en `Writeback`. Como se vio en clases, en este caso **se asume que el `Writeback` alcanza a actualizar los registros para que en `Instruction Decode` se seleccionen los valores nuevos**.
- Se modifica el caso de *hazard* de datos en el que hay una dependencia entre la primera y segunda instrucción de un programa con lectura de memoria: como ahora `Execute` y `Memory` se ejecutan en el mismo ciclo, el valor obtenido de memoria **alcanza a ser transmitido por *forwarding* a la siguiente instrucción sin tener que esperar un ciclo**. Por lo tanto, se elimina la lógica de *stalling* y la *data hazard unit* que la implementa.

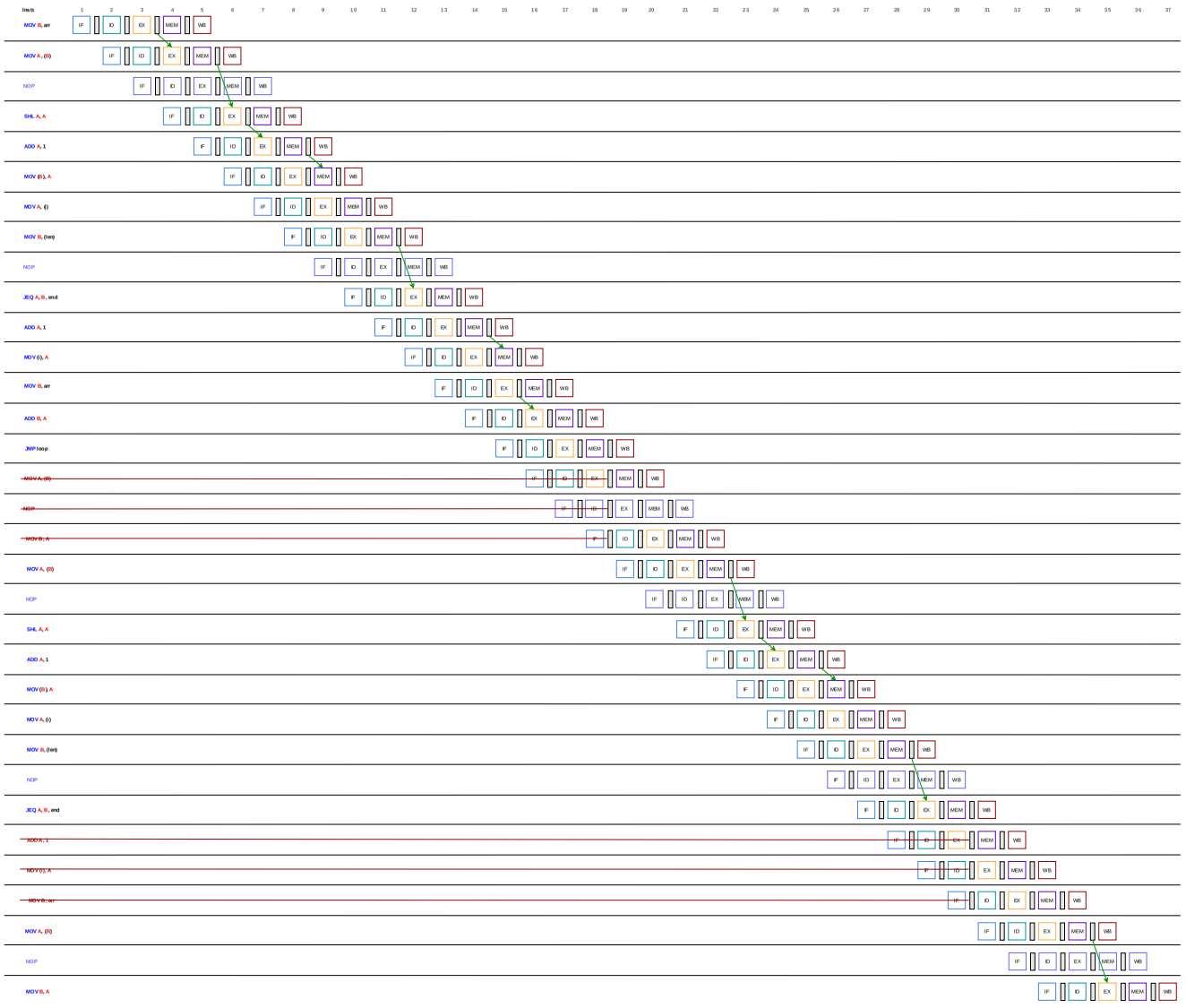
Por cada implicancia, se otorga **1 pto.** si es correcta y está debidamente explicada.

(b) (4 ptos.) A partir del siguiente programa del computador básico con *pipeline*:

```
DATA:
  arr 1
      2
      3
  len 1
  i   0
CODE:
  MOV B, arr
  loop:
    MOV A, (B)
    SHL A, A
    ADD A, 1
    MOV (B), A
    MOV A, (i)
    MOV B, (len)
    JEQ A, B, end
    ADD A, 1
    MOV (i), A
    MOV B, arr
    ADD B, A
    JMP loop
    end:
      MOV A, (B)
      MOV B, A
```

Determine el número de ciclos que demora su ejecución detallando en un diagrama los estados del *pipeline* por instrucción. Asuma que el manejo de *stalling* es por *software* a través de la instrucción `NOP` y que la unidad predictora de saltos asume que estos nunca se realizan, independiente de que estos sean condicionales o incondicionales. Indique en el diagrama adjunto al enunciado cuando ocurre *forwarding* (con flechas desde el registro hacia la etapa que corresponda), *stalling* y *flushing* (con tachado en las instrucciones *flusheadas*).

Solución: En la siguiente plana, se muestra el *pipeline* resultante de la ejecución del programa.


Instr.
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
MOV B, arr
MOV A, (0)
NOP
SHL A, A
ADD A, 1
MOV (B), A
MOV A, (i)
MOV B, (sm)
NOP
JEQ A, B, end
ADD A, 1
MOV (i), A
MOV B, arr
ADD B, A
JMP loop
MOV A, (0)
NOP
MOV B, A
MOV A, (0)
NOP
SHL A, A
ADD A, 1
MOV (B), A
MOV A, (i)
MOV B, (sm)
NOP
JEQ A, B, end
ADD A, 1
MOV (i), A
MOV B, arr
MOV A, (0)
NOP
MOV B, A
IF ID EX MEM WB

Del *pipeline* anterior se deduce que el programa termina de correr después de 37 ciclos. Se otorga puntaje de la siguiente forma: **0.3 ptos.** por cada *stalling* correctamente detectado (**1.25 ptos.** por todas las detecciones); **0.1 ptos.** por cada *forwarding* correctamente detectado (**1.2 ptos.** por todas las detecciones); **0.6 ptos.** por cada *flushing* correctamente detectado (**1.2 ptos.** por todas las detecciones); y **0.35 ptos.** por la deducción correcta de la cantidad de ciclos en la que corre el programa.

Se otorga el puntaje por cantidad de ciclos solo si se obtiene el valor correcto por la ejecución correcta del programa y **no por accidente por errores de ejecución**.

Si no se agrega correctamente un *stall*, pero se aplican bien los *forwardings*, se otorga puntaje en dicho criterio. No obstante, si existen uno o más *forwardings* mal aplicados, se descuentan **0.2 ptos.**

No hay descuento si se incluyen las instrucciones de escritura de memoria correspondientes al segmento `DATA`, pero solo si la ejecución de estas se describe de forma correcta.

Nota: Si bien el arreglo posee tres elementos, la configuración de `len` con valor igual a 1 genera que `loop` se itere de forma completa una sola vez y que se termine el programa en la segunda ejecución de `JEQ A, B, end`.