Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

IIC2343 – Arquitectura de Computadores

Ayudantía 6 – Solución propuesta

Profesor: Yadran Francisco Eterovic Solano

Ayudante: Germán Leandro Contreras Sagredo (glcontreras@uc.cl)

Nota al lector

El título dice 'solución propuesta' por una razón bien sencilla: Estos ejercicios pueden tener más de un desarrollo correcto. Lo que se pretende hacer aquí es mostrar un camino a la solución, sin excluir la posibilidad de rutas alternativas igual de correctas.

Preguntas

1. a. **(Examen - I/2013)** Indique qué capacidades/intrucciones gana y pierde el computador básico al agregar soporte para paralelismo a nivel de instrucción.

   Lo más importante que gana el computador básico, que se deduce del tema en sí, es el poder de ejecutar múltiples instrucciones en paralelo. Sin embargo:

   - Se pierden los llamados a subrutinas. Como el registro `SP` se elimina y no existe una conexión del `PC` a la memoria, las subrutinas no se pueden implementar.
   - No se pueden hacer instrucciones de salto que impliquen el uso de los bit de estado N, C, O (pues se eliminan). Notar que además, como se realiza la comparación $A - B$ y el salto en un mismo ciclo, la `ALU` se conecta directamente a un nuevo elemento: la unidad de salto (separando esa funcionalidad de la unidad de control). Por esto, se elimina además el registro *Status*, pues ya no es necesario.
   - Ya no se pueden usar datos de la memoria directamente como parámetros en la `ALU`, ya que se elimina la conexión entre el Mux `B` y la salida de memoria. Además, tampoco se puede enviar directamente el resultado de la `ALU` a la memoria. Lo único permitido es la transferencia de memoria desde y hacia los registros `A` y `B`.

   b. **(I3 - II/2016)** En caso que la única solución para solucionar un *hazard* de datos sea introducir una burbuja, ¿cuál es el momento más adecuado para hacerlo?

   El momento más adecuado para hacerlo es en la etapa ID (*Instruction Decode*), ya que se desea poder identificar el *hazard* lo más pronto posible para perder menos ciclos. La etapa ID es la más temprana en la que se puede detectar, por lo que al identificar el *hazard* se querrá pausar un ciclo de forma inmediata.

c. **(Examen - I/2013)** Un computador con microarquitectura avanzada posee un *pipeline* de 30 etapas. ¿Cuál es el elemento de *hardware*, sin contar los registros entre etapas, que tiene el mayor impacto (positivo y negativo) en el rendimiento? Justifique su respuesta.

Al poseer una cantidad considerable de etapas, el elemento más crítico, a la larga, será la unidad predictora de saltos. Si se implementa mal (*i.e.* posee un bajo porcentaje de acierto en las predicciones), se podrían perder muchos ciclos a medida que se avanza en las etapas.

d. **(Examen - II/2015)** Un computador x86 monoprocesador posee un pipeline de 5 etapas que se ejecutan en el siguiente orden:

a) *Fetch*: Obtiene desde la memoria el *opcode* de la instrucción a ejecutar.
b) *Decode*: Decodifica el *opcode*, enviando las señales correspondientes a cada componente.
c) *Read*: Lee desde registros y memoria los datos requeridos para ejecutar la operación.
d) *Execute*: Ejecuta la operación aritmética/lógica de la instrucción usando la `ALU`.
e) *Write*: Almacena en registros o memoria el resultado de la operación aritmética/lógica.

Además de los mecanismos tradicionales para combatir *hazards* de datos y de control, el computador evita *hazards* estructurales de acceso a memoria entre las etapas *Fetch*, *Read* y *Write*, al utilizar una memoria RAM que permite realizar de manera simultánea tres solicitudes distintas. A pesar de esto, es posible generar un *hazard* estructural de ejecución entre las etapas *Read* y *Execute*, cuando se procesa una instrucción del tipo `MOV Reg1,[Reg2+offset]`. ¿Cómo es posible evitar este *hazard*?

Se puede ver que el *hazard* se produce al necesitar utilizar la `ALU` dos veces dentro de la misma instrucción (una para poder almacenar en `Reg1` un dato, y otra para obtener `Reg2+offset`). Una solución **para este caso en particular** sería añadir un sumador en la etapa *Read*. De esta forma, al buscar datos en memoria, `Reg2` y `offset` podrían pasar por el sumador y el resultado ser utilizado para obtener el dato correcto a almacenar en `Reg1`.

2. a. **(I3 - II/2016)** En promedio, ¿cuántos ciclos por salto pierde un computador con un *pipeline* de 12 etapas, si su unidad predictora acierta el 75 % de las veces? Asuma que los saltos se realizan en la penúltima etapa.

Si los saltos se realizan en la etapa 11 (correspondiente a la penúltima etapa), entonces son 10 las instrucciones que se podrían perder por una mala predicción. Asumiendo un ciclo por instrucción, se tiene entonces que se pierden 10 ciclos por un salto mal realizado. Finalmente, si se falla en el 25 % de las veces, se pierde un total de $10 * 0{,}25 = 2{,}5$ ciclos por salto.

b. **(I3 - I/2016)** Estime el tiempo de ejecución de un programa que toma N instrucciones en el computador básico con *pipeline*, en base a las siguientes condiciones:

- El 50 % de las instrucciones realizan lecturas o escrituras en memoria.
- El 10 % de las instrucciones son de salto y en el 25 % de estas, el salto finalmente se realiza.
- En el 50 % de las ocasiones, el dato obtenido desde la memoria debe ser utilizado en la instrucción siguiente.

Cualquier supuesto sobre la solución debe quedar claramente explicado.

De partida, con N instrucciones, se tendrá un total de $N + 4$ ciclos si no se consideran las condiciones anteriores. Luego, se van añadiendo más iteraciones según los siguientes criterios:

- Considerando el *pipeline* visto en clases, tenemos que se agregan 3 ciclos por cada salto realizado (correspondientes a los que se les hizo *flush*) si se asume una unidad predictora que siempre opta por no saltar. Como se tiene un 10 % de instrucciones de salto y un 25 % de estos se realiza, entonces se añade un total de $0{,}1 * 0{,}25 * 3 * N = 0{,}075N$ ciclos al total.
- Como el 50 % de las instrucciones acceden a memoria y en el 50 % de estos casos el dato obtenido de memoria se usa en la siguiente instrucción, en $0{,}5 * 0{,}5 * N = 0{,}25N$ instrucciones se necesita hacer *stalling* (por lo que se pierde una cantidad de ciclos igual a la cantidad de veces que se realiza esta espera, pues solo se pierde un ciclo por burbuja insertada).

Finalmente, tenemos que el tiempo de ejecución final es:

$$t \times ((N + 4) + (0{,}075N) + (0{,}25N)) = 4 \times t + 1{,}325N \times t$$

Siendo t el tiempo que toma la ejecución de un ciclo.

3. **(I3 - II/2016)** Considere el siguiente código escrito para el assembly del computador básico con *pipeline*:

```
ADD A,B
MOV (var0),A
MOV (var1),B
```

a. Indique con un diagrama los *hazards* que se generan al ejecutar el código anterior.

El *hazard* se genera con las líneas 1 y 2 del código, donde se escribe en la dirección `var0` lo que está contenido en el registro `A`, cuando este no ha sido modificado aún por la etapa WB. El diagrama, entonces, queda como sigue:

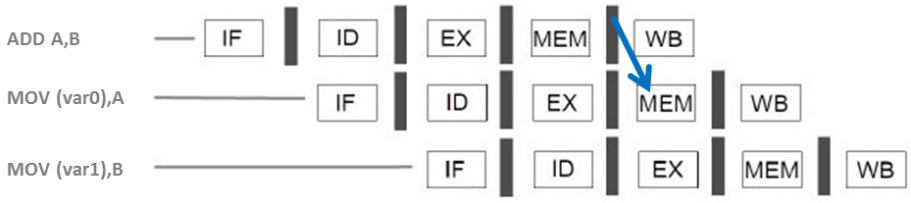

Figura 1: *Hazard* de datos por las instrucciones 1 y 2.

b. Indique qué sucede al intercambiar el orden de la segunda y tercera línea del código. ¿Cambia el resultado final? En caso de existir *hazards*, indíquelos con un diagrama.

El *hazard* ahora se genera con las líneas 1 y 3 del código, donde nuevamente se escribe en la dirección `var0` lo que está contenido en el registro `A`, cuando este no ha sido modificado aún en la etapa WB. El diagrama, entonces, queda como sigue:

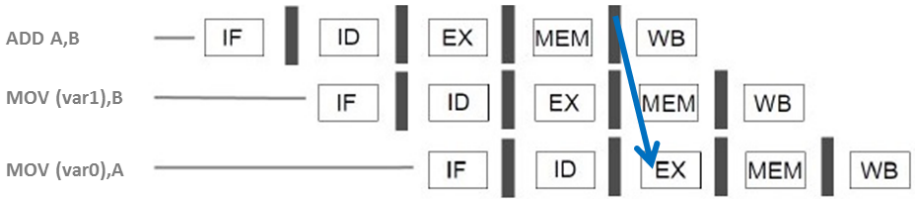

Figura 2: *Hazard* de datos por las instrucciones 1 y 3.

c. Para los dos casos anteriores, en caso de existir *hazards*, indique cómo los detectan las *forwarding units* correspondientes, especificando las señales de control participantes.

   I. Para identificar el primer caso, como se puede ver a partir del diagrama, la *forward unit* 2 revisa que se cumpla el siguiente criterio: `MEM/WB.LoadA == 1` (es decir, se realizará una escritura en el registro `A` por parte de la primera instrucción), `EX/MEM.W == 1` (es decir, se realizará una escritura en memoria por parte de la segunda instrucción) y `EX/MEM.MemIn == A` (es decir, se ingresará a memoria lo que está en el registro `A`). Con esto, la *forward unit* 2 sabe que debe propagar la salida del Mux `RegIn` al Mux `FwDin` activando la señal *ForwardDin* para que se escoja el resultado de la instrucción anterior y se escriba de forma efectiva en memoria.

   II. Para identificar el segundo caso, como se puede ver a partir del diagrama, la *forward unit* 1 revisa que se cumpla el siguiente criterio: `MEM/WB.LoadA == 1` (es decir, se realizará una escritura en el registro `A` por parte de la primera instrucción), `ID/EX.W == 1` (es decir, se realizará una escritura en memoria por parte de la tercera instrucción) y `ID/EX.MemIn == A` (es decir, se ingresará a memoria lo que está en el registro `A`). Con esto, la *forward unit* 1 sabe que debe propagar el resultado de la instrucción al Mux `FwA` activando la señal *Forward A* para que se escoja la señal `ALU` (que se encuentra almacenada en MEM/WB).

4. **(I3 - I/2016)** 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 no ocurre. Indique en el diagrama cuando ocurre *forwarding*, *stalling* y *flushing*.

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

Si el *forwarding* se marca con flechas azules, el *stalling* con instrucciones `NOP` y el *flushing* con líneas rojas que marcan el código que no se ejecutará, se tiene el siguiente diagrama:

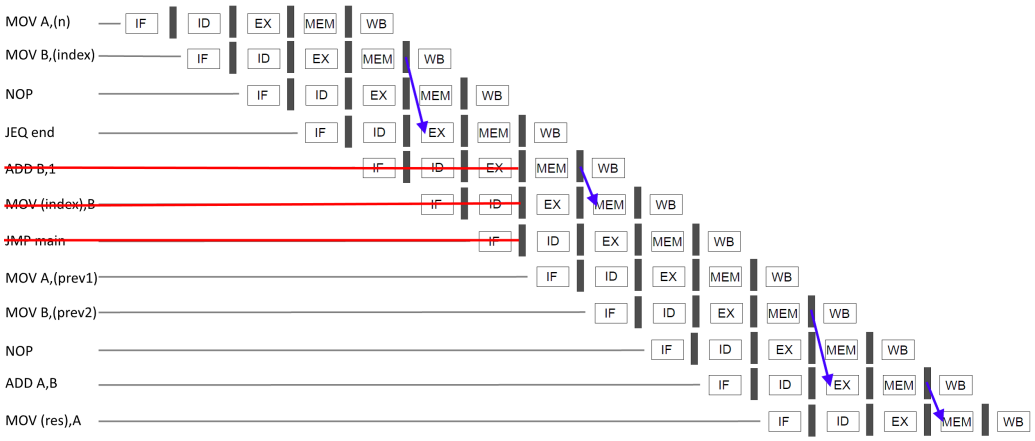

Figura 3: Diagrama final de la ejecución del programa, con un total de 16 ciclos.