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

IIC2343 – Arquitectura de Computadores

Ayudantía 9 – Solución propuesta

Profesores: Hans-Albert Löbel Díaz, Jurgen Dieter Heysen Palacios

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. **(I3 - I/2018)** Al hacer un computador con *pipeline*, por ejemplo en el caso del computador básico, ¿qué componentes se agregan?

   Los elementos que se agregan son esencialmente cuatro:

   - Registros de separación entre cada fase.
   - Unidades de control de *hazard*.
   - Unidades de *forwarding*.
   - Unidad de salto.

   b. **(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`.

c. **(I3 - I/2018)** ¿Por qué se permite hacer *forwarding* en algunos casos en un *pipeline* en lugar de hacer *stalling* de la `CPU`?

   Debido a que es posible determinar que, en esos casos, se requiere el dato de forma rápida y es mejor permitir la ejecución en lugar de parar la `CPU` por uno o más ciclos, influyendo positivamente en la rapidez de ejecución de los programas dentro de esta arquitectura.

d. **(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.

e. **(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.

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:

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

3. a. **(Examen - I/2018)** Considere la arquitectura del computador básico con *pipeline*. Una de las grandes desventajas que posee es el hecho de que si la unidad predictora de saltos se equivoca, se pierden tres ciclos, lo que impacta considerablemente el tiempo de ejecución de los programas. ¿Cómo modificaría esta arquitectura para poder perder un ciclo menos por predicción de salto? Puede hacer un diagrama o detallar su implementación.

   Bastaría con la adición de un sustractor que tenga de entradas los valores de las salidas de `Mux FwA` y `Mux FwB` dentro de la etapa `EX`, además de trasladar a esta sección la unidad de salto y la unidad de *hazard* de control. De esta forma, en la etapa `EX` la unidad de *hazard* de control puede determinar si es necesario hacer un *flush* o no, desechando dos ciclos en vez de tres. Notar que esto hace innecesario el uso de la *flag* `Z` de la `ALU`, ya que pierde su propósito.

b. **(Examen - I/2018)** Suponga que tiene una empresa que se encuentra desarrollando un dispositivo relativamente simple, que no ejecuta programas de más de 10 instrucciones. Uno de los desarrolladores sugiere que, en pos de la eficiencia, lo diseñen con un *pipeline* de 10 etapas para poder paralelizar las ejecuciones. ¿Es esta una decisión conveniente tomando en cuenta el tiempo de ejecución de los programas? Justifique su respuesta.

   No. Al existir el proceso de *filling* (llenado de las etapas del *pipeline*), recién en el décimo ciclo todas las instrucciones estarían ejecutándose en una etapa del *pipeline*, tomando un total de 19 ciclos de ejecución. Si se tuviera una arquitectura como la del computador simple, el programa tendría una ejecución de 10 ciclos, además de tener una menor limitante en lo que respecta a las instrucciones. En resumen, la decisión perjudicaría la *performance* del dispositivo. **Importante:** Se asume que la diferencia del tiempo de ejecución **de un ciclo** en el dispositivo con y sin *pipeline* no es significativa, validando el análisis anterior.

c. **(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`.

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:

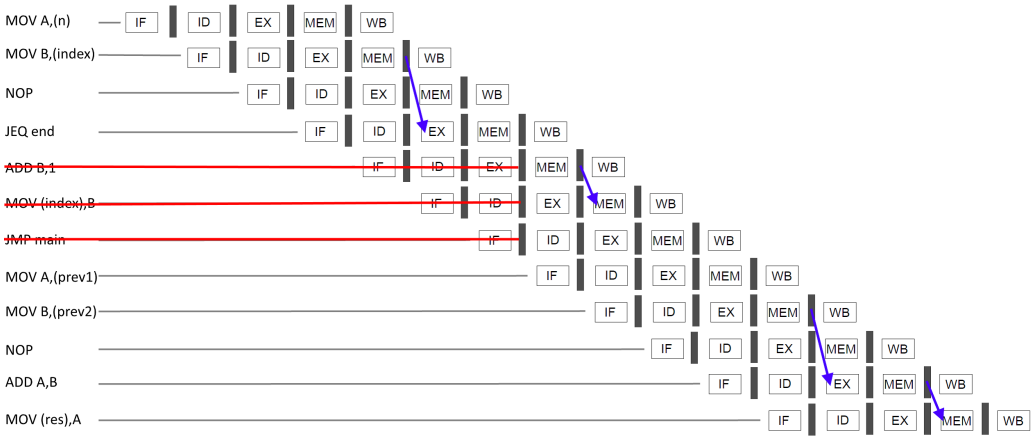

Figura 1: Diagrama final de la ejecución del programa, con un total de 16 ciclos.