Departamento de Cs. e Ingeniería de la Computación
Universidad Nacional del Sur

Arquitectura de Computadoras

Trabajo Práctico N^{o} 7
Pipeline de Instrucciones
Primer Cuatrimestre de 2010

Ejercicios

1. La ganancia de velocidad en un procesador con pipeline de instrucciones está acotada por la siguiente expresión:

$$\textit{Pipeline Speedup} = \frac{\textit{Ideal } \mathbf{CPI} \times \textit{Pipeline Depth}}{\textit{Ideal } \mathbf{CPI} + \textit{Pipeline Stall Cycles}}$$

   a) Si se asume que libre de problemas de recursos se alcanza un **CPI** *ideal* de 1 y que un 30 % de las instrucciones son de referencia a memoria, ¿cual sería la mayor ganancia de velocidad alcanzable al adoptar un esquema de *cache independiente para datos e instrucciones* (acceso a cache de un ciclo), respecto a tener una *cache unificada*?

   b) Asumiendo que la frecuencias de *hazards* y requerimientos de *stalls* entre una instrucción y sus sucesoras son de 20 % para dos ciclos de stall y de 5 % para un ciclo de stall, calcular la mayor ganancia de velocidad alcanzable en este escenario.

2. Un cierto procesador cuenta con un pipeline del tipo visto en la teoría con cinco etapas, a saber: FETCH, DECODE, EXECUTE, MEMORY y WRITE-BACK, cuyo banco de registros se lo accede dos veces por ciclo, para escritura en la primera mitad y para lectura en la segunda. Sobre este procesador, se desea ejecutar la siguiente secuencia de instrucciones:

$$\begin{aligned}
I_1: &\quad R_1 \leftarrow R_2 \times R_3 \\
I_2: &\quad R_6 \leftarrow R_4 \times R_2 \\
I_3: &\quad R_5 \leftarrow M[R_1] \\
I_4: &\quad R_4 \leftarrow R_5 + R_6 \\
I_5: &\quad R_2 \leftarrow R_4 \times R_3 \\
I_6: &\quad R_3 \leftarrow R_5 + R_6 \\
I_7: &\quad R_4 \leftarrow R_1 + R_2
\end{aligned}$$

   Esquematizar en un diagrama de Gantt el solapamiento entre instrucciones, indicando en cada uno de los siguientes escenarios los *hazards* observados y el origen, de haberlos, de los ciclos de **stall** del pipeline y/o de los ciclos de espera (**wait**) a nivel de instrucciones individuales. Por último, indicar también la cantidad de ciclos requeridos para ejecutar la totalidad del fragmento de código en cuestión.

   a) Se dispone de dos unidades funcionales, una de suma de un ciclo y otra de producto de cuatro ciclos, con *buffer de entrada* y sin *forwarding*.

b) Idem anterior pero disponiendo de tantos niveles de *forwarding* entre unidades funcionales (ya sea **ALU** o de referencia a memoria) como resulte necesario, haciendo explícitos los *bypasses* realizados.

c) Idem escenario *b* pero implementando *D–virtuales* (según se vio en teoría) para el circuito multiplicador.

d) Idem escenario *b* pero estructurando la unidad multiplicadora en pipeline de cuatro etapas, lo cual permitirá inicial un producto en cada ciclo (de no media dependencias de datos, claro está).

Obs: De haber colisión en la etapa de WRITE-BACK, esto no debe ser considerado un *hazard*, ya que sólo se está demorando la escritura. Es decir, especificar simplemente cuántos pórticos de escritura debería disponer el banco de registros para evitar esta demora.

3. Realizar la evaluación de los tiempos de ejecución de la siguiente secuencia de instrucciones, asumiendo un pipeline de cinco etapas con dos unidades funcionales: una para el producto, la cual demora cuatro ciclos y está estructurada en pipeline, y la otra para la suma de un único ciclo:

$$\begin{aligned} I_1: &\quad R_1 \leftarrow R_2 \times R_3 \\ I_2: &\quad R_2 \leftarrow R_5 + R_5 \\ I_3: &\quad R_6 \leftarrow R_2 \times R_1 \\ I_4: &\quad R_1 \leftarrow R_1 \times R_2 \\ I_5: &\quad R_2 \leftarrow R_3 + R_5 \end{aligned}$$

a) Ejecución en orden, sin forwarding.

b) Ejecución en orden, con forwarding (hacerlo explícito).

c) Con forwarding y ejecución fuera de orden (*despacho condicional*). Determinar de acuerdo a la dinámica impuesta por la secuencia de instrucciones cuál será la cantidad de *estaciones de reservación* asociadas al sumador y al multiplicador para que no exista stall por dependencia de recursos.

Obs: Al igual que antes, de haber colisión en la etapa WRITE-BACK se deberá indicar el número de pórticos para que ésta desaparezca. Se asume resuelto el conflicto **WAR**.

4. Recordando que como medida de desempeño se tiene:

   - Tiempo de ejecución para una determinada tarea (**latencia**), o bien
   - Cantidad de trabajo realizado en la unidad de tiempo (**throughput**).

   y que se establece que una computadora A resulta $n\,\%$ más veloz que otra computadora B para una determinada aplicación si:

$$\frac{\textit{Tiempo Ejecución de } B}{\textit{Tiempo Ejecución de } A} = \frac{100+n}{100}$$

   En este contexto, se desea cotejar el desempeño de dos procesadores de distintas arquitecturas en la ejecución de un determinado programa. Uno de ellos se basa en la arquitectura **Intel** funciona a 400 Mhz y puede ejecutar hasta cuatro instrucciones por ciclo de reloj en un pipeline de doce etapas, pero debido a dependencias entre las instrucciones y el

miss en cache, el rendimiento para el programa en cuestión presenta un **IPC** de 1,6. El otro procesador, basado en la arquitectura **Alpha** tiene un tiempo de ciclo de 1,5 ns, un pipeline de nueve etapas y para este programa alcanza un **CPI** de 0,8. Además, dada la naturaleza antagónica de las arquitecturas (RISC vs. CISC), el segundo procesador necesita una 40 % más de instrucciones para realizar la misma tarea.

a) ¿Cuál es la *latencia mínima* en nanosegundos de una instrucción para ambos procesadores?

b) ¿Cada cuantos nanosegundos promedio se ejecuta una instrucción en cada procesador? ¿Guarda alguna relación con la latencia mínima?

c) Determinar qué procesador es el más rápido para la aplicación considerada y en qué porcentaje.

5. Un cierto procesador implementa un pipeline con las siguientes características: **CPI** ideal de 1 y carga con un porcentaje de instrucciones de que pueden modificar el flujo de control de un 17 %, de los cuales el 13 % son saltos condicionales y el 4 % restante incondicionales. Tener en cuenta que en esta arquitectura la frecuencia con que los saltos condicionales son tomados es en promedio del 62 %.

   Realizar el análisis del **CPI** efectivo y de la ganancia de velocidad real de este pipeline en cada uno de los siguientes escenarios:

   a) Con stall de pipeline (esto es, sin predicción).

   b) Predicción *branch tomado*.

   c) Predicción *branch no tomado*.

   d) Con *branch retardado* (considerando que el slot de retardo se pueda llenar con éxito en el 50 % de los casos).

   Para los primeros tres escenarios asumir que el branch se completa en la etapa MEMORY, pero a los efectos de beneficiarnos con el esquema predicción tomado, asumir que el nuevo PC_{target} se calcula en la etapa DECODE, optimización parcial. Para el último caso, utilizar optimización total, esto es, se resuelve en etapa DECODE tanto el PC_{target} como la evaluación de la condición.

   Obs: Recordar los factores que afectan la ganancia de velocidad del pipeline, a saber:

$$\textit{Pipeline Speedup} = \frac{\textbf{CPI} \times \textit{Pipeline Depth}}{\textbf{CPI} + \textit{Pipeline Stall Cycles from Branches}}$$

$$\textit{Pipeline Stall Cycles from Branches} = \textit{Branch Frequency} \times \textit{Branch Penalty}$$

6. Un cierto procesador cuenta con un *pipeline de instrucciones* de cinco etapas. Cuenta con dos unidades funcionales, una de suma en un ciclo y otra de producto en tres ciclos. Tanto datos como instrucciones siempre producen *hit* en cache, con un *hit time* de un ciclo y se admiten dos accesos al banco de registro por ciclo (primera mitad escritura, segunda mitad lectura). Finalmente, los conflictos de control son resueltos mediante *stalls*.

   Este procesador debe ejecutar la siguiente secuencia de instrucciones:

```
I1 :          ADD   R1, R2, R3
I2 :          MULT  R3, R2, R1
I3 :          MULT  R4, R4, R4
I4 :          JUMP  label
I5 :          MULT  R5, R5, R4
I6 : label:   ADD   R6, R6, R4
I7 :          MULT  R7, R5, R6
I8 :          ADD   R8, R5, R6
```

En este contexto, bosquejar un *diagrama de Gantt* detallando la evolución de las instrucciones en el pipeline y determinar los distintos **CPI** promedios alcanzables en cada uno de los siguientes escenarios, indicando claramente los *stalls* y los *waits* que se produzcan y todos los *forwardings* que se requieran (según corresponda):

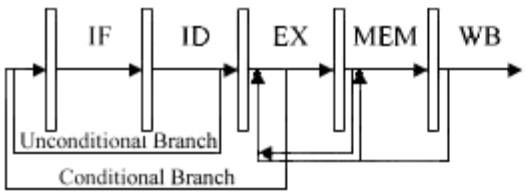
a) Pipeline básico, *sin forwarding*.

b) Pipeline *con forwarding* y *D-Virtuales*.

c) Pipeline avanzado, *con forwarding* y *con pipeline lineal en las unidades funcionales*.

Obs: En caso de producirse un doble requerimiento de escritura en un ciclo, esto **no constituye conflicto que detenga al pipeline**; más aun, se puede asumir que no se detiene la ejecución por disponer de múltiples pórticos de escritura en el banco de registros. Finalmente, se asume que los saltos incondicionales **se resuelven en su totalidad de forma óptima**, esto es, durante la etapa de decodificación.

7. Considerando un procesador con una organización pipeline de cinco etapas como se muestra a continuación. Se indican sobre el mismo la organización de las etapas y caminos de control y de datos entre las mismas. Las instrucciones se ejecutan siempre en orden y el estado del procesador también se modifica en orden. Se supone además que no se producen miss en cache, ni de datos ni de instrucciones. En cuanto a los saltos, se continúa con el fetch secuencialmente y se modifica el control cuando esto sea posible, esto es *predicción no tomado*.



Instrucción	Resol.	Semántica
BR label	ID	Salto incondicional a "label"
B*cc* Rx,label	EX	Salto condicional, si *cc*(Rx)
OP Rx,Ry,Rz	EX	(Rx OP Ry) → Rz
LD Rx,off(Ry)	MEM	Memory load in Rx
ST off(Ry),Rx	MEM	Memory store from Rx

Se muestra además el repertorio de instrucciones. Los saltos condicionales se resuelven al final de la tercer etapa y los incondicionales en la segunda. Las operaciones aritméticas al final de la tercera. Las de acceso a memoria terminan al final de la cuarta etapa. Las operaciones no tienen que esperar que sus operandos estén almacenados en registros destino, sino que pueden obtener el valor por caminos de bypass, justo en el ciclo siguiente a que se resuelva la instrucción anterior (esto es, mediante *forwarding*).

a) ¿Qué tipos de conflictos pueden producirse en este procesador? ¿Dónde se deberían detectar y qué habría que hacer en cada caso?

b) Indicar el número de ciclos de penalización que produce cada conflicto.

c) Suponiendo que sobre este procesador se ejecuta el programa que se indica a continuación, según la siguiente secuencia:

$$1, 2, 3, 4, 10, 5, 6, 7, 8, 9, 10, 11$$

En este contexto, realizar una tabla con la evolución en la ejecución del programa hasta completar la instrucción número 11, indicando las penalizaciones que se produzcan.

```
1. 100 ADDQI R31,1,R3    ; R3←R31+1
2. 104 LDA   R8,V(R31)   ; R8←R31+V   load address
                         ; does not access memory!!
3. 108 LDQ   R2,0(R8)    ; R2←MEMORY(R8)
4. 112 BR    +20         ; branch to instr. 10
5. 116 LDQ   R1,8(R8)    ; R1←MEMORY(R8+8)
6. 120 ADDQ  R2,R1,R2    ; R2←R2+R1
7. 124 STQ   R2,0(R8)    ; MEMORY(R8)←R2
8. 128 ADDQI R8,8,R8     ; R8←R8+8
9. 132 SUBQI R3,1,R3     ; R3←R3-1
10.136 BNE   R3,-24      ; if R3!=0 branch to instr. 5
11.140 LDA   R8,X(R28)   ; R8←R28+X   load address
```

cycle	*1*	*2*	*3*	*4*	*5*	*6*
IF	1	2	3	4	.	.
ID		1	2	3	.	.
EX			1	2	.	.
MEM				1	2	.
WB					1	2

Referencias

[HP96] Hennessy, J., and Patterson, D. *Computer Architecture*, second ed. Morgan Kaufmann, 1996.