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

IIC2343 – Arquitectura de Computadores (I/2014)

Examen

- Duración 2:15 horas (08:45 - 11:00)
- Conteste una pregunta por hoja. Escriba su nombre en cada hoja de respuestas, hágalo ahora!
- **No hay preguntas**. Si tiene alguna duda escriba los supuestos necesarios junto con su respuesta.

Pregunta 1 (1.5 puntos)

A continuación se muestran dos algoritmos escritos en pseudo-código, cada uno implementado de dos maneras diferentes. Para medir el desempeño de éstos se tiene una función `tic()` que marca el inicio del "cronómetro" y una función `toc()` que marca el fin. Para cada uno de los ejemplos indique qué código corre más rápido (o bien si no hay diferencias), considerando solamente el código que está entre `tic()` y `toc()`. Justifique.

Caso 1 (0.5 puntos)

```
int matrix() {
  byte mat[N][M] =
    random_matrix(N,M);
  int sum = 0;
  tic();
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
      sum += mat[i][j];
    }
  }
  toc();
  return sum;
}
```

```
int matrix() {
  byte mat[N][M] =
     random_matrix(N,M);
  int sum = 0;
  tic();
  for (int j = 0; j < M; j++) {
    for (int i = 0; i < N; i++) {
      sum += mat[i][j];
    }
  }
  toc();
  return sum;
}
```

Caso 2 (1 punto)

```
int sum() {
  byte arr[N] = random_array(N);
  int sum = 0;
  tic();
  for (int i = 0; i < N; i++) {
    if (arr[i] < 128) {
      sum++;
    }
  }
  toc();
  return sum;
}
```

```
int sum() {
  byte arr[N] = random_array(N);
  arr = sort(arr);
  int sum = 0;
  tic();
  for (int i = 0; i < N; i++) {
    if (arr[i] < 128) {
      sum++;
    }
  }
  toc();
  return sum;
}
```

Respuesta

Caso 1 En este caso corre más rápido el caso de la izquierda, ya que el código va accediendo a lugares contiguos de memoria, optimizando el uso de la caché. La implementación de la derecha debe ir "saltando" por diferentes espacios de memoria, logrando un hit rate mucho más bajo.

Caso 2 Corre más rápido el de la derecha. Como el arreglo está ordenado, la CPU predice de menor manera el branch a seguir debido al `if`. En la realidad, la CPU va ajustando dinámicamente el criterio de branch prediction, lo cual hace que el código de la derecha corra de manera más eficiente.

Pregunta 2 (1.5 puntos)

Se tiene una caché de 16 entradas 2-way associative con 4 palabras por bloque, utilizada para un espacio de memoria de 32 palabras. Indique lo siguiente:

1. Cómo se dividen los bits de la dirección de memoria de una palabra para ser utilizados en la información de la tabla de caché? **(0.5 puntos)**

   Respuesta Los 5 bits de la dirección ($2^5 = 32$), distribuidos como $b_4b_3b_2b_1b_0$ se pueden dividir de la siguiente manera:

   - b_4: Conjunto
   - b_3, b_2: Tag
   - b_1, b_0: Ubicación de la palabra en el bloque

   El orden de los bits utilizados no importa (y por ende los accesos en la parte 2 podrían cambiar), pero los dos bits de tag al igual que los dos bits de ubicación de la palabra en el bloque deben ser contiguos.

2. Utilizando la tabla diseñada en el punto anterior, indique el estado de la memoria caché después de acceder a las siguientes direcciones: **0, 1, 3, 4, 9**

   Use LFU para reemplazar y LRU para desempatar. **(1 punto)**

Respuesta Los primeros tres accesos son al mismo bloque de memoria. Pondremos este bloque en el primer bloque disponible del conjunto 0:

Conjunto	Índice de bloque	Ubic. Palabra	Bit validez	Tag	# Accesos	Tiempo	Dato
0	0	00	1	00	3	1	mem[0]
		01	1	00	3	1	mem[1]
		10	1	00	3	1	mem[2]
		11	1	00	3	1	mem[3]
	1	00	0				
		01	0				
		10	0				
		11	0				
1	0	00	0				
		01	0				
		10	0				
		11	0				
	1	00	0				
		01	0				
		10	0				
		11	0				

Para el acceso a la memoria 4 debemos utilizar el segundo bloque disponible:

Conjunto	Índice de bloque	Ubic. Palabra	Bit validez	Tag	# Accesos	Tiempo	Dato
0	0	00	1	00	3	2	mem[0]
		01	1	00	3	2	mem[1]
		10	1	00	3	2	mem[2]
		11	1	00	3	2	mem[3]
	1	00	1	01	1	1	mem[4]
		01	1	01	1	1	mem[5]
		10	1	01	1	1	mem[6]
		11	1	01	1	1	mem[7]
1	0	00	0				
		01	0				
		10	0				
		11	0				
	1	00	0				
		01	0				
		10	0				
		11	0				

Para el último acceso reemplazamos el bloque 1 del conjunto 0, debido a que ha sido accedido menos veces:

Conjunto	Índice de bloque	Ubic. Palabra	Bit validez	Tag	# Accesos	Tiempo	Dato
0	0	00	1	00	3	3	mem[0]
		01	1	00	3	3	mem[1]
		10	1	00	3	3	mem[2]
		11	1	00	3	3	mem[3]
	1	00	1	10	1	1	mem[8]
		01	1	10	1	1	mem[9]
		10	1	10	1	1	mem[10]
		11	1	10	1	1	mem[11]
1	0	00	0				
		01	0				
		10	0				
		11	0				
	1	00	0				
		01	0				
		10	0				
		11	0				

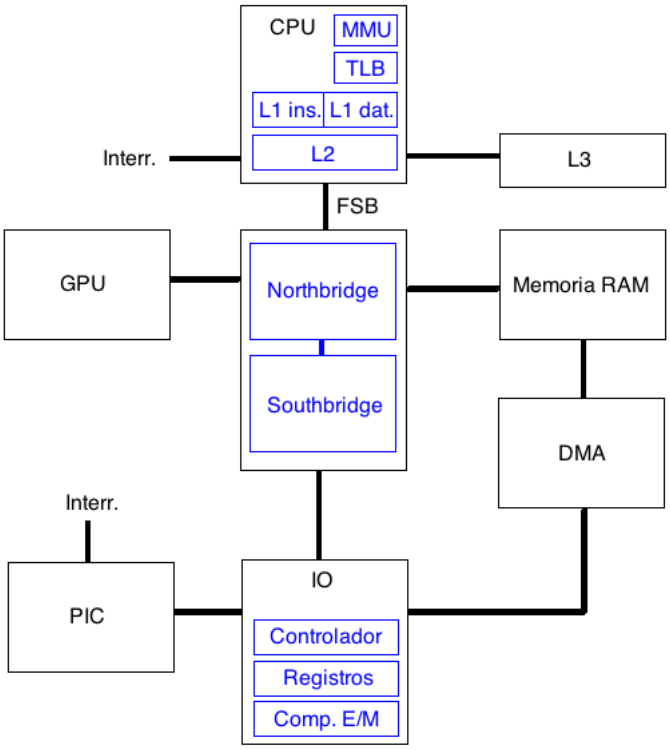
No fue necesario usar LRU para desempatar.

Pregunta 3 (1.5 puntos)

Haga un diagrama de la arquitectura de buses de un computador Intel. Debe incluir los siguientes componentes:

- CPU
- Componentes de la arquitectura de buses intel
- Dispositivos IO (Detalle su estructura interna)
- Interrupciones
- DMA
- Esquema de memoria tipo Von Neumann
- 3 niveles de Caché
- GPU
- Memoria Virtual (MMU + TLB)

Respuesta Los componentes se muestran en el siguiente diagrama:



Pregunta 4 (1.5 puntos)

Responda las siguientes preguntas:

1. De qué sirve tener timers e interrupciones de timers en la CPU? **(0.5 puntos)**

   Respuesta Es crucial ya que sin estos timers el sistema operativo no puede quitar el control a un programa de manera forzada. Dado que el sistema operativo puede configurar estos timers para interrumpir a la CPU en un tiempo posterior, es que éste tiene la seguridad que adelante la CPU, mediante la interrupción del timer, interrumpirá al proceso volviendo el control al sistema operativo.

2. Por qué pueden haber problemas de coherencia de caché en una CPU de 1 solo core? **(0.5 puntos)**

   Respuesta Pueden haber problemas si un dispositivo de IO escribe en memoria usando DMA. Al escribir con DMA la información no pasa por la CPU, por lo cual se podría escribir en memoria un dato que existe en caché sin que la CPU sepa sobre el cambio, produciéndose problemas de coherencia de caché.

3. Explique qué es out-of-order excecution. **(0.5 puntos)**

Respuesta Es una técnica en computación que permite ejecutar instrucciones en assembly de manera desordenada sin afectar el propósito original del programa a ejecutar. Por ejemplo, teniendo las instrucciones:

```
add A, 1
sub B, 2
add A, B
```

puedo intercambiar las instrucciones 1 y 2 sin afectar el estado final del programa. Esto sirve para hacer más eficiente el pipeline, considerando que pueden haber accesos a memoria principal, dependencia de datos u otras demoras que pueden ser compensadas haciendo out-of-order execution.