Aspectos Avanzados de Arquitectura de Computadoras y Taller de Arquitectura de Computadoras

Prueba escrita correspondiente al curso 2014

- **Indique su nombre completo y número de cédula en cada hoja.**
- **Numere todas las hojas e indique el total de hojas en la primera. Escriba las hojas de un solo lado.**
- **Solo se responderán dudas de letras. No se responderán preguntas en los últimos 30 minutos de la prueba.**
- **La prueba es individual y sin material. Duración de la prueba : 3 hs.**
- **Todas las preguntas tienen el mismo puntaje y el mínimo de aprobación es de un 60%.**
- **Justifique todas sus respuestas.**

Pregunta 1

Considere un procesador superescalar que funciona con política *Out-of-Order Issue Out-of- Order Completion.* Responda:

a) ¿Qué es la Ventana de Instrucciones (Instruction Window)? Explique cómo se utiliza en el pipeline superescalar y cómo ayuda a la explotación del ILP (*Instruction Level Paralelism*).

b) Explique qué es la etapa de commit, y por qué es necesaria.

Solución Pregunta 1

a)

La ventana de instrucciones es una estructura de datos de hardware que se ubica en la etapa issue (entre la etapa decode y execute), su objetivo es alojar las instrucciones que ya fueron decodificadas y que están a la espera de una unidad funcional libre, o de que se generen los operandos origen. Con esto se colabora a la explotación del ILP porque instrucciones posteriores en el orden lógico del programa pueden ser despachadas hacia una unidad funcional antes que sus predecesoras lo hayan hecho (Out-of-Order Issue), permitiendo un mejor uso de las unidades funcionales.

b)

La etapa Commit es una etapa que aparece en procesadores superescalares con out-of-order completion, como estrategia para prevenir algunos problemas que pueden surgir por la ejecución fuera de orden. Cuando una instrucción pasa por la etapa commit significa que sus resultados han impactado lógicamente y ya no es revertible. Las instrucciones pasan por la etapa de commit *en orden* (son reordenadas si es necesario), con el objetivo de evitar que instrucciones especulativas que no deberían ser ejecutadas impacten, y para mantener excepciones precisas.

Pregunta 2

a) Deduzca la expresión de la generalización de la Ley de Amdahl en términos de la fracción afectada y la aceleración (speedup) afectada por una determinada modificación en la organización de la CPU.

b) Exprese el límite de la expresión anterior cuando la aceleración afectada tiende a infinito, y de una interpretación de la misma en términos de la utilidad de introducir mejoras a un diseño.

Solución Pregunta 2

a)

$$T_{old} = T_{no\ afectado} + T_{afectado\ old}$$

$$T_{new} = T_{no\ afectado} + T_{afectado\ new}$$

$$T_{new} = T_{no\ afectado} + \frac{T_{afectado\ old}}{speedup_{afectado}}, \text{ donde } speedup_{afectado} = \frac{T_{afectado\ old}}{T_{afectado\ new}}$$

$$T_{new} = \left(1 - Fracción_{afectada}\right) T_{old} + \frac{Fracción_{afectada}\, T_{old}}{speedup_{afectado}}$$

$$speedup_{total} = \frac{T_{old}}{T_{new}} = \frac{1}{\left(1 - Fracción_{afectada}\right) + \frac{Fracción_{afectada}}{speedup_{afectado}}}$$

Observación: $Fracción_{afectada}$ es con respecto a T_{old}

b) Si la aceleración de la parte afectada tiende a infinito, la aceleración total tiende a 1/(1-F). La interpretación de esta expresión es que la parte afectada tiene que ser suficientemente grande para tener un efecto importante en la aceleración. Si se acelera mucho un aspecto poco significativo, las ganancias son pocas, lo que en economía se denomina “diminishing returns”.

Pregunta 3

¿Cuál es la función de la Unidad de control en la organización de la CPU? Indique alternativas para implementarla y explique en qué consiste cada una de ellas.

Solución Pregunta 3

La Unidad de Control debe controlar la secuencia de pasos discretos que son necesarios para decodificar y ejecutar las instrucciones (implementar el ciclo de instrucción).

Las opciones son:

Control Cableado: Se implementa como un circuito secuencial, usando compuertas y componentes (NAND, NOR, flip-flops, contadores, etc.). Son rápidos pero pueden ser inflexibles , ya que si se necesita agregar instrucciones extra, el circuito debe rediseñarse .

Control microprogramado: Solución en software del control de la máquina. Para cada "macroinstrucción" existe una secuencia específica de microinstrucciones que la implementa. El código elemental necesario para implementar el ciclo de instrucción de la CPU se almacena en memoria de microprograma de tipo ROM (firmware), por lo que suele no ser necesario un cambio de hardware para implementar nuevas instrucciones.

Ejercicio 1

Dado un conjunto de ***n*** de objetos, una *combinación* C_r^n es una selección de ***r*** objetos sin importar el orden en que se escojan. Las combinaciones se pueden calcular utilizando la siguiente recurrencia:

$$C_r^n = \begin{cases} 1 & si\ r = 0\ o\ r = n \\ n & si\ r = 1 \\ C_r^{n-1} + C_{r-1}^{n-1} & en\ otro\ caso \end{cases}$$

Se pide:
Compile la función ***C*** en 8086, pasando parámetros y resultado en el stack. Deben conservarse los valores de todos los registros y retirarse los parámetros del stack al retornar.

Solución Ejercicio 1

Llamada:

```
push n
push r
call c
pop result
```

Procedimiento

```
c proc
push bp ; guardo el registro bp
mov bp, sp ; apunto con bp al tope del stack
push ax ; conservo registros para luego restaurarlos
push bx
push cx
push dx
mov ax, [bp+6] ; obtengo n
mov bx, [bp+4] ; obtengo r
cmp bx, 0 ; comparo r con 0
je ret1
cmp bx, 1 ; comparo r con 1
je retN
```

```
cmp bx, ax ; comparo r con n
je ret1
dec ax ; decremento en uno a n
push ax ; coloco parametros en el stack
push bx ;
call c ; realizo la llamada recursiva con n-1 y r
pop cx ; obtengo el resultado
dec bx ; decremento en uno a r
push ax ; coloco parametros en el stack
push bx
call c ; realizo la llamada recursiva con n-1 y r-1
pop dx ; obtengo resultado
add cx, dx
fin: mov [bp+6], cx ; muevo el resultado a memoria
mov ax, [bp+2] ; obtengo ip
mov [bp+4], ax ; almaceno ip en el lugar correcto del stack
pop dx ; restauro registros
pop cx
pop bx
pop ax
pop bp
add sp, 2 ; dejo el sp apuntado a ip
ret
ret1: mov cx, 1 ; asigno el resultado del paso base
jmp fin
retN: mov cx, ax ; asigno el resultado del paso base
jmp fin
c endp
```

Ejercicio 2

Considere un CPU de 16 bits que emite direcciones de 24 bits, operando con la siguiente jerarquía de memoria direccionable por bytes:

- Un cache de datos de primer nivel de 8KB, con organización de correspondencia asociativa por conjuntos de dos vías y política de remplazo FIFO. La misma cuenta con un tamaño de línea de 32 bytes.
- Memoria RAM.

Sea el siguiente código que calcula la suma de tres arreglos:

```
short arreglo_a[2048];
short arreglo_b[2048];
short arreglo_c[2048];
short arreglo_resultado[2048];
// inicialización de los arreglos
...
// fin inicialización de los arreglos
for (int i = 0; i < 2048; i++){
      arreglo_resultado[i] = arreglo_a[i] + arreglo_b[i] + arreglo_c[i];
}
```

a) Indique cómo interpreta el cache las direcciones de memoria. Halle el valor de todos los parámetros relevantes.

b) Suponiendo que la cache se vacía luego de la inicialización de los arreglos, que los arreglos se posicionan consecutivos en memoria a partir de la dirección 0x800000, que un short se compila a 16 bits, calcule el *miss_rate* del código brindado.

c) Se considera la técnica *merge arrays* para tratar de explotar mejor la localidad espacial:

```
typedef struct nodoSuma{
      short a;
      short b;
      short c;
      short resultado;
}
struct nodoSuma mergeArreglos[2048];
// inicialización de la estructura
```

```
...
// fin inicialización de la estructura
for (int i = 0; i < 2048; i++){
      mergeArreglos[i].resultado = mergeArreglos[i].a +
                                   mergeArreglos[i].b  +
                                   mergeArreglos[i].c
}
```

Calcule el *miss_rate* para este programa manteniendo las suposiciones de la parte anterior. Compare este resultado con el de la parte anterior.

Solución Ejercicio 2

a) Por ser una cache asociativa por 2 vías, la dirección se interpreta como:

dirección: $_{23}$ | tag | set | byte |$_{0}$

Como la cache de datos tiene 8KB y sus palabras son de 32 bytes, se deduce que la misma dispone de 256 líneas, y como éstas se asocian de a 2 vías, la cache se divide en 128 sets. Dado esto, el campo 'set' de la dirección ocupa 7 bits (ya que $2^7 = 128$), el campo byte ocupa 5 bits (ya que $2^5 = 32$) y el campo 'tag' ocupa los restantes 12 bits.

b) Como el primer arreglo se ubica en la posición 0x800000 y ocupa 2048 * 2 bytes = 4KB, el segundo arreglo se ubica a partir de la posición 0x801000, por tanto el tercero se ubica a partir de la dirección 0x802000 y el arreglo resultado se ubica a partir de 0x803000.

El primer acceso es a la dirección 0x801000, la cual da miss por estar la cache vacía y se carga el bloque de tag 0x801 en la línea 1 del set 0.

El segundo acceso es a la dirección 0x802000, al cual da miss por no encontrarse el bloque de tag 0x802 en ninguna de las líneas del set 0. Se va a buscar dicho bloque a memoria y se ubica en la segunda línea del tag.

El tercer acceso es a la dirección 0x803000, al cual da miss por no encontrarse el bloque de tag 0x803 en ninguno de las líneas del set 0. Se va a buscar dicho bloque a memoria y se ubica en la primer línea del tag, desplazando la línea de tag 0x801.

De ahí en adelante, cada acceso resulta en miss pues aún si dicho bloque fue cargado anteriormente, fue desplazado por los posteriores accesos. Por lo tanto el miss rate es 1.

c) En esta parte, por la disposición en memoria de la estructura, los accesos a memoria son a palabras de 16 bits (2 bytes) consecutivas de memoria. Dado esto, los misses ocurrirán en cada primera referencia a un bloque de memoria y los accesos a las siguientes palabras del bloque serán

hit. Como cada bloque tiene 32 bytes (16 palabras de 2 bytes), 1 de cada 16 accesos será miss y el resto será hit.

Por lo tanto el miss rate es 1/16.