# Complemento de Arquitectura de Computadoras

Solución Examen 14 de Febrero de 2017

**(ref: scac20170214.odt)**

- **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 letra. No se responderán preguntas en los últimos 30 minutos de la prueba.**
- **La prueba es individual y sin material. Duración de la prueba : 3 horas.**
- **El puntaje mínimo de aprobación es de 60 puntos.**
- **Justifique todas sus respuestas.**

## *Pregunta 1 (10 puntos)*

En un procesador superescalar con ejecución fuera de orden, describa y comente cuál es la funcionalidad de:

1. La ventana de instrucciones.
2. La etapa Commit.

## *Pregunta 2 (10 puntos)*

**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.

## *Pregunta 3 (12 puntos)*

1. ¿De qué forma alivia a la CPU el uso de DMA para transferir datos desde E/S a memoria?
2. Describa el fenómeno de robo de ciclos de la CPU por el DMA y cómo afecta su respuesta anterior.

## *Pregunta 4 (8 puntos)*

En una arquitectura con pipeline, describa la técnica de forwarding. Indique qué tipo de hazards soluciona y por qué.

## *Ejercicio 1 (30 puntos)*

El algoritmo *run lenght encoding* es un algoritmo de compresión básico, el cual parte de un arreglo de caracteres y genera otro arreglo donde se indica, de forma ordenada cuántas veces aparece el caracter de forma consecutiva.
Ejemplo:

```
[A,A,A,D,D,D,C,C]          ->          [(3A),(3D),(2C)]
[A,A,A,C]                  ->          [(3A),(1C)]
[C,A,C]                    ->          [(1C),(1A),(1C)]
```

Considere el siguiente tipo de datos:

```
typedef struct codChar{
      char caracter;
      short cant;
};
```

a) Implementar en un lenguaje de alto nivel (preferentemente C), la función *run length encoding*, la cual tiene le siguiente cabezal:

```
short runLengthEncoding(char* arreglo, short lengthIn, codChar* codif);
```

El parámetro *lengthIn* indica el largo de la variable arreglo y la función devuelve el largo de la variable *codif*. Suponga que hay suficiente memoria reservada en las variables 'arreglo' y 'codif'.

b) Compilar a assembler 8086. Los parámetros *arreglo* y *codif* se pasan en SI y DI respectivamente

como desplazamientos con respecto al segmento ES y el parámetro *lengthIn* se pasa en el registro CX. El resultado se devuelve en el registro AX.

## *c) Solución*

b) Implementar en un lenguaje de alto nivel (preferentemente C), la función *run length encoding*, la cual tiene le siguiente cabezal:

```
short runLengthEncoding(char* arreglo, short lengthIn, codChar* codif){
 short i;
 short lengthOut = 0;
 for (i = 0; i < lengthIn; i++){
       char charActual = arreglo[i];
       short cantIguales = 0;
       while((i + cantIguales < lengthIn) &&
                   (arreglo[i + cantIguales] == charActual)){
             cantIguales++;
       }
       codif[lengthOut].caracter = charActual;
       codif[lengthOut].cant = cantIguales;
       lengthOut++;
       i += cantIguales;
 }
 lengthOut--;  // Corrijo: size = ultimaPosicion - 1
 return lengthOut;
}
```

```
 runLenghtEncoding PROC
       XOR BX, BX;             ; i = 0
       XOR BP, BP;             ; lengthOut = 0
 for:
       CMP BX, CX              ; i < lenghtIn ?
       JGE finfor
       MOV AL, ES:[SI + BX]
       XOR DX, DX              ; cantIguales
 while:
       ADD BX, DX              ; bx = i + cantIguales
       CMP BX, CX
       JGE finwhile
       CMP ES:[SI + BX], AL
       SUB BX, DX              ; bx = i
       JNE finwhile
       INC DX
       JMP while
 finwhile:
       SUB BX, DX
       ; Convierto BP (índice) en puntero
       PUSH BP                       ; guardo indice, después lo preciso
       PUSH AX
       MOV AX, BP
       ADD BP, AX
       ADD BP, AX                    ; BX = BX * 3
       POP AX
       MOV ES:[DI + BP], AL
       MOV ES:[DI + BP + 1], DX
       ADD BX, DX                    ; i += cantIguales
       POP BP                        ; recupero índice
       INC BP
finfor:
       DEC BP
       MOV AX, BP
       RET
 ENDP
```

## *Ejercicio 2 (30 puntos)*

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 bloque 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;
      int 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*

***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 segundo se ubica a partir de la dirección 0x802000, el tercero a partir de la dirección 0x803000 y el arreglo resultado se ubica a partir de 0x804000.

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 el set 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.