Universidad Tecnológica Nacional
Facultad Regional Buenos Aires
Departamento de Ingeniería Electrónica

Cátedra: Técnicas Digitales III - Plan 95A

Guía de Trabajos Prácticos

Ciclo Lectivo 2013

Índice

Índice de figuras

Índice de cuadros

Parte I.
Generalidades

1. Introducción y Régimen de aprobación

La presente guía de Trabajos Prácticos tiene por objeto llevar a la práctica los contenidos vistos en las clases teóricas. De este modo se espera una realimentación entre la comprensión de los diferentes conceptos teóricos y su aplicación práctica efectiva, desarrollando en el alumno un enfoque metodológico y sistémico que le permita resolver problemas de Ingeniería utilizando como herramientas los diferentes componentes digitales, subsistemas y sistemas abordados a lo largo del ciclo lectivo.

El grado de complejidad irá creciendo a través de los diferentes ejercicios planteados para cada Unidad Temática.

Cada alumno deberá presentar aquellos ejercicios que lleven la indicación ***Entrega Obligatoria***. La entrega de cada ejercicio se efectuará sin excepciones en las fechas estipuladas en el cronograma de clase que se entregará en la primera clase del ciclo lectivo. Los calendarios de entrega de los prácticos obligatorios, estarán diseñados para que todos los Trabajos Prácticos correspondientes a los contenidos que se incluyen en cada parcial sean revisados por los docentes auxiliares antes del examen. De este modo los alumnos tendrán una devolución con las correcciones de los errores detectados, como forma de realimentación necesaria para el examen parcial.

La no entrega de un ejercicio en la fecha establecida equivale a considerar al alumno o al grupo Ausente en ese práctico, considerándose en consecuencia, No Aprobado dicho práctico. En el caso de esta guía de Trabajos Prácticos la aprobación del 100 % de los estipulados de ***Entrega Obligatoria***, es una de las condiciones necesarias para la aprobación por promoción o firma de Trabajos Prácticos.

2. Formato de presentación

- Los archivos fuente deben tener en todos los casos los comentarios necesarios para clarificar su lectura.
- Como encabezado del programa, debe haber un comentario que explique claramente la tarea que realiza, y las instrucciones detalladas (comandos) para su compilación y enlace o “linkeo”.
- Cada subrutina/función, debe contar con un encabezado describiendo la operación que realiza, los parámetros que espera como entrada, y los resultados que debe presentar, indicando formato y método de entrega.
- Los comentarios deben estar escritos preferentemente en formato de alguna herramienta de generación de documentación adecuada, por ejemplo “***Doxygen***”.
- Se debe proveer como entregable: Los archivos fuente, el archivo “`makefile`” o “`shell script`”con la secuencia de comandos necesarios para construir el programa mediante la simple orden *make* o *./[shellscript]*, y el archivo de configuración de la herramienta de documentación empleada, en el caso de Doxygen, el archivo Doxyfile.

3. Procedimiento para la entrega de los Trabajos Prácticos

La entrega de los trabajos prácticos se realizará en primer lugar en un sistema de control de versiones del tipo SVN administrado por la cátedra. El alumno utilizará para acceder a dicho repositorio el mismo usuario y clave que utiliza para ingresar en el Sistema de Gestión Electrónica (SGE) del Departamento.

El alumno subirá a este repositorio los ejercicios prácticos intermedios que lo conducirán al entregable, con la frecuencia en que vaya dedicándose a las tareas de ejercitación previstas en la asignatura. Por cada ejercicio de la guía deberá tener una carpeta separada dentro de su directorio.

La versión final del trabajo práctico se compondrá de los programas fuentes necesarios, el "`makefile`" que permita su compilación y un archivo de texto plano "***readme***" con las instrucciones adicionales que el alumno considere pertinente para la ejecución, en "***Bochs***", de su programa. Además, cada archivo fuente deberá estar correctamente documentado. Para ello se recomienda la herramienta "***Doxygen***".

> Importante: NO SUBIR LA DOCUMENTACIÓN. Solo es necesario el ***Doxyfile*** con el cual los docentes armarán su copia de documentación en forma local. "***Doxygen***" genera un arbol de directorios sumamente profundo lo cual hace colapsar al software de versiones cuando se quiere acceder al repositorio SVN desde un cliente.

La no presencia de la versión final completa del Trabajo Práctico en la fecha estipulada en el repositorio indicado se considerará ausente.

Solo bajo prueba fundada de indisponibilidad del servicio de internet de la Facultad se aceptarán entregas vía e-mail, siempre que los docentes lo hayan autorizado expresamente por comunicación a través de la lista de correo de los diferentes cursos.

Parte II.
Trabajos Prácticos

1. Modo Protegido

El presente trabajo práctico está pensado de manera incremental, de modo que cada ejercicio tomará como base el precedente, añadiéndole el código necesario para implementar las funcionalidades y modificaciones requeridas.
Recomendaciones:
Utilice una estructura de directorios que asigne cada ejercicio a una carpeta. Una vez finalizado y verificado el correcto funcionamiento de un ejercicio, copie los ficheros de código fuente a la carpeta asignada al siguiente ejercicio y a partir de este comience a implementar lo requerido. Utilice el número de ejercicio como parte del nombre del archivo fuente, por ejemplo para el ejercicio 3.1 tp_3_1.
Todos los ejemplos deben arrancar mediante el mecanismo de boot establecido por la Cátedra.

1.1. Inicialización minimalista en modo protegido 32bits

Escriba un programa que cumpla los siguientes requerimientos:

a) Poner el procesador a trabajar en modo protegido.

b) Leer la dirección de E/S 0x60, a la espera del número 0x01 (código de Scan de la tecla ESC).

c) Finalizar su ejecución quedando en estado ***halted*** en forma permanente.

El programa no debe tener definida una GDT.
¿Como debe estar el Flag de Interrupciones del procesador? ¿porque?

1.2. Inclusión de una GDT y manejo básico de segmentos

Agregar al programa del Ejercicio 1.1 los siguientes ítems:

a) Agregar una GDT con dos segmentos de DPL=00, Base0x0 y 4Gbytes de tamaño, uno de datos y otro de código.

b) Definir una pila dentro del segmento de datos e inicializar el par de registros SS:ESP adecuadamente.

c) Poner la pantalla de video modo texto en video inverso.

d) Leer la dirección de E/S 0x60, a la espera del número 0x01.

e) Finalizar su ejecución mediante quedando en estado ***halted*** en forma permanente.

El acceso a la pantalla de video se efectúa escribiendo en un área de memoria de 4000 bytes (**no 4 Kbytes**) denominada Video RAM Buffer que se muestra en la Figura 1.

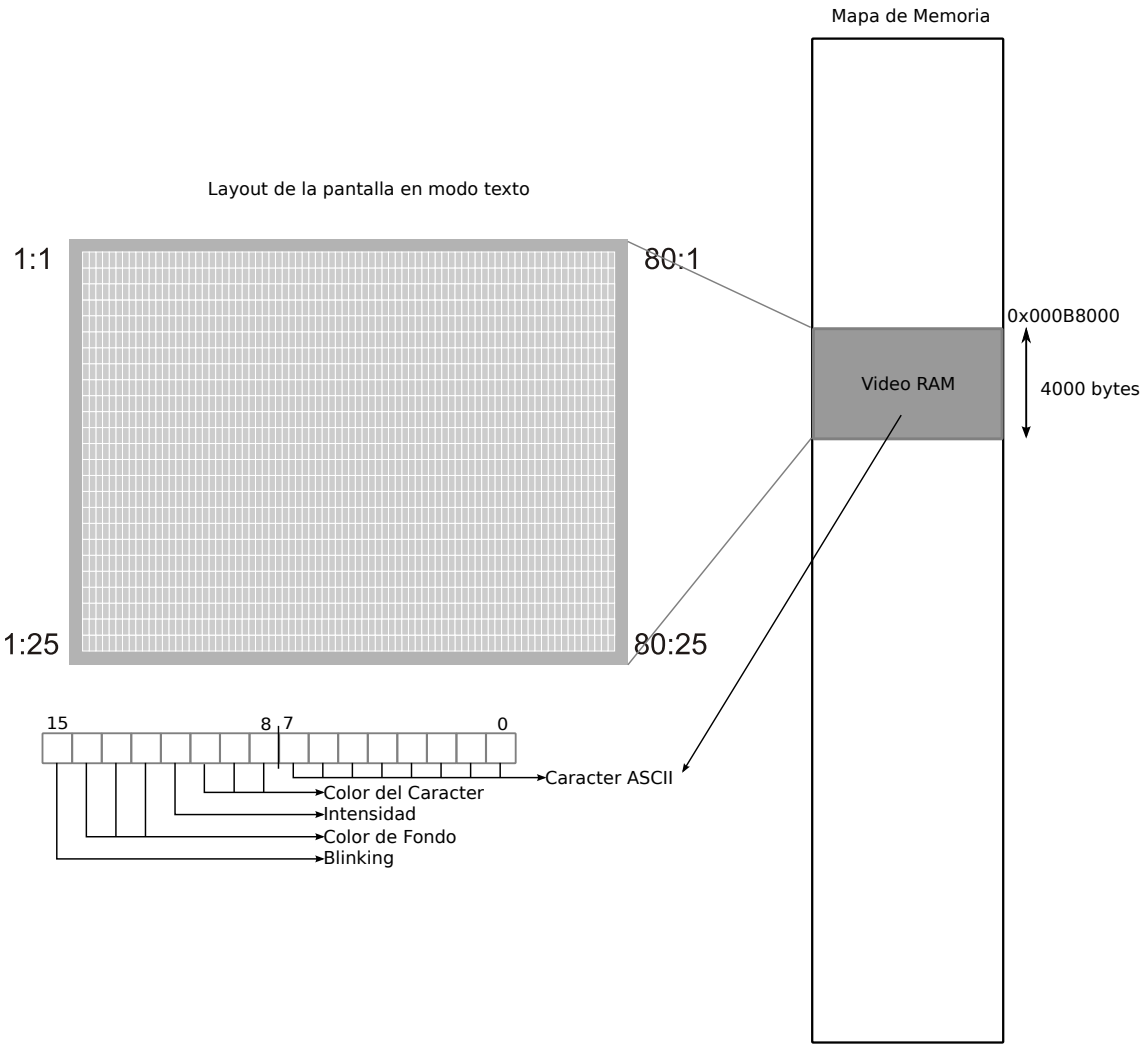

Fig. 1: Mapeo de la pantalla en la Video RAM y formato de cada word

1.3. Habilitando Paginación con prudencia: Identity Mapping

Tome el programa desarrollado en el Ejercicio 1.2, y active paginación, armando previamente las tablas de traducción adecuadamente para que cada dirección lineal se traduzca en la misma dirección física. Trabaje solamente con el primer Mbyte de RAM.

Sugerencia:
Explorar la directiva `%rep` y `%assign` del NASM como herramienta para replicar estructuras de datos repetitivas.

1.4. Habilitando PAE. Siempre en Identity Mapping

Modifique el programa 1.3, de modo de habilitar PAE, manteniendo el mapeo del primer Mbyte de RAM en identity mapping.

1.5. Inicialización en 64 bits. Modo Compatibilidad

Tome el programa del Ejercicio 1.4, y modifique lo necesario para ingresar al modo IA-32e. Se desea mantener en principio los segmentos de 32 bits.

1.6. Manejo sencillo de Excepciones e interrupciones en 64bits

En base al ejercicio 1.5, agregue una tabla IDT capaz de manejar todas las interrupciones y excepciones del procesador mas las que estén reservadas hasta el tipo 0x2F. Además deberá:

a) Configurar el PIC maestro y esclavo de manera que utilicen el rango de tipos de interrupción 0x20-0x27 y 0x28-0x2F respectivamente.

b) Inicializar el registro de máscaras de modo que estén deshabilitadas todas las interrupciones de hardware, en ambos PIC's.

c) Implementar cada una las rutinas de atención a excepción, de manera que impriman en pantalla el número de excepción generada y finalicen mediante la instrucción "hlt".

1.7. Interrupciones de Hardware

Tome el Ejercicio 1.6 y agregue los siguientes componentes:

a) Habilite *Gate A20* para acceder a la memoria por encima del 1er. Mbyte de RAM.

b) Extienda el mecanismo de traducción de páginas para que abarque sin generar excepción #PF los primeros 2 Mbytes de RAM.

c) Efectúe la prueba de acceso con una dirección superior al 1er. Mbyte. Note que si Gate A20 no está correctamente habilitada el dato se escribe en la posición de memoria cuyos 20 bits menos significativos coinciden con la dirección utilizada para el acceso.

d) Reemplace la secuencia de lectura de la dirección de E/S 0x60, por un controlador de IRQ_1 (teclado) que detecte que se ha presionado la tecla ESC, tras lo cual se debe establecer el procesador en estado ***halted*** en forma permanente.

1.8. Manejo avanzado de Interrupciones

A partir del Ejercicio 1.7, ampliar la rutina de atención de IRQ_1 para detectar las combinaciones de teclas que se indican a continuación, las cuales deben efectuar operaciones que generen la excepción que se indica para cada una:

Combinación de teclas	Excepción a generar
CTRL+1	Interrupción Tipo 3: #BP Breackpoint Trap
CTRL+2	Interrupción Tipo 7: #DF Double Fault
CTRL+3	Interrupción Tipo 13: #GP General Protection
CTRL+4	Interrupción Tipo 14: #PF Page Fault

Tab. 1: Excepciones a provocar por cada combinación de teclas

Importante:
No es válido activar las rutinas de atención de las excepciones mediante la instrucción **INT *n***, siendo ***n*** el tipo de la interrupción solicitada. La generación de la excepción debe ser producto de la ejecución de una o mas instrucciones que generen las condiciones establecidas para la excepción (bse).

1.9. Manejo de Excepciones avanzado *Entrega Obligatoria*

A partir del **Ejercicio 1.7**, desarrollar un sistema que inicialmente solo contenga, en la estructura de traducción de direcciones lineales a físicas, las tablas mínimas necesarias para su carga

y ejecución en memoria.
Se pide adicionalmente los siguientes requerimientos:

a) Implementar en el flujo principal del sistema, una secuencia pseudo-aleatoria de acceso a memoria de datos. A tal fin se recomienda utilizar la instrucción ***RDRAND*** o bien el registro del PIT #0 que controla el timer tick, para generar el desplazamiento dentro del rango de direcciones asignadas a los datos. Se pretende acceder a direcciones lineales generadas al azar independientemente de que éstas hayan sido o no asignadas al programa.

b) Implementar un controlador de excepción de #PF que cumpla con la siguientes especificaciones:

   Ubicar las páginas correspondientes a la dirección virtual requerida por el sistema, en páginas físicas consecutivas a partir del 1er. Mbyte de memoria física, **independientemente del valor de la dirección lineal generada por la Unidad de Segmentación**.

   Crear una nueva página cada vez que la aplicación intente acceder a una dirección sin presencia de página física en RAM

c) El sistema se debe establecer en modo ***halted*** en forma permanente, cuando se agote la memoria física existente.

1.10. Niveles de privilegio

Tomar el código del Ejercicio 1.9 y modificar/agregar los siguientes ítems:

a) Agregaren la GDT un segmento de datos y otro de código de 64 bits ambos con PL =11.

b) Generar en el segmento de datos una pila para el PL=11.

c) Modificar los atributos de los descriptores de segmento con PL=00 (*kernel*) para que pasen a ser de 64 bits.

d) Modificar la rutina que accede a memoria de modo que se encuentre en un segmento de código y solo se utilice para páginas de (*kernel*).

e) Modificar el privilegio de la página que contiene dicha rutina, adecuándolo para que pueda ejecutarse sin generar una excepción #PFa

f) La función que genera la semilla pseudo aleatoria, debe ser accedida como un servicio. Debe ubicarla en el segmento de código y página acorde con su privilegio.

g) Diseñe un mecanismo de acceso a la función aleatoria.

1.11. Conmutación de tareas

Al código del Ejercicio 1.10, es necesario incorporarle una capacidad mínima de administrar tareas. Para ello se requiere, agregarle las siguientes prestaciones:

a) Modificar el valor del temporizador 0 del PIT, para que genere una interrupción cada 1 mseg, aproximadamente.

b) Implementar un controlador para la interrupción 32 (IRQ_0 timer tick), de manera que en cada interrupción se conmute de tarea.

c) El mecanismo de conmutación de tareas debe seguir los lineamientos indicados en: http://wiki.electron.frba.utn.edu.ar/doku.php?id=td3:switchto.

Por ahora debe utilizarse el código que invoca a la función aleatoria del *kernel*, como tarea. A los efectos de este ejercicio basta con generar dos instancias diferentes de este mismo código como tareas.

Nota:
En esta instancia del desarrollo puede utilizar un CR3 único para ambas tareas, si esto le genera demasiada complejidad. El uso de un CR3 diferente para cada tarea se requiere en el Ejercicio 1.12.

1.12. Paginación por cada tarea

En base al código del Ejercicio 1.11 implementar una estructura de paginación independiente para cada tarea, a fin de que ambas instancias posean un adecuado marco de protección de sus áreas de memoria. Si esta funcionalidad la implementó en el Ejercicio 1.11, continúe con el Ejercicio siguiente.

1.13. Mejoras en el *scheduler*. Lista de Tareas

Tomar el código del Ejercicio 1.12, y agregar los siguientes ítems:

a) a. Manejar las tareas mediante una lista doblemente enlazada de modo de independizar el código del *scheduler* de la cantidad de tareas a ejecutar.

b) Asignar a cada tarea un valor de prioridad. Dicho valor equivale a la cantidad de ticks, durante los cuales el *scheduler* no conmuta la tarea en curso. Los valores de prioridad van de 1 a 20.

1.14. Otras tareas

Tomar el Ejercicio 1.13 e incluir el código necesario para implementar:

a) Funciones del kernel que se invoque según la siguiente descripción:
`char *fecha(void)`, devuelve la fecha del sistema en formato “aa/mm/dd”.
`char *hora(void)` , devuelve la hora del sistema en formato “hh/mm/ss”.
`long jiffies(void)`, devuelve la cantidad de milisegundos efectiva de uso de la CPU consumidos desde su inicio por la tarea invocante.
`void msleep(int)`, duerme al proceso (no es conmutado por el scheduler) durante la cantidad de mseg que se indica en su argumento. Durante este lapso no se incrementará el contador de mseg. de consumo de CPU de la tarea. `int random ()`, es la función desarrollada desde el principio `void puts (int, int, char *)`, Imprime a partir de la posición [fila,columna] dada por los dos primeros argumentos, la cadena ASCIIZ cuyo puntero se pasa como tercer argumento.

b) Reemplazar las dos tareas que se tienen hasta ahora e implementar en su lugar las siguientes, las cuales se resuelven utilizando las funciones del kernel descriptas en el párrafo anterior:

Fecha . Imprime en la ventana inferior izquierda de la Figura 2, a razón de un renglón por cada una: Fecha,y Tiempo consumido de CPU. La función invocará a la función random (), con la cual obtendrá un número pseudoaleatorio, que trasladado a mili segundos mediante algún factor de escala y redondeo, determinará el valor a pasarle a msleep ().

Hora . Imprime en la ventana inferior izquierda de la Figura 2, a razón de un renglón por cada una: Horra,y Tiempo consumido de CPU. La función invocará a la función random (), con la cual obtendrá un número pseudoaleatorio, que trasladado a mili segundos mediante algún factor de escala y redondeo, determinará el valor a pasarle a msleep ().

De este modo se espera que cada tarea vaya cambiando su carga de CPU de manera aleatoria.

Fig. 2: Formato de las ventanas de cada tarea

1.15. Tareas con uso de instrucciones SIMD ***Entrega Obligatoria***

Al sistema del Ejercicio 1.14, agregar una tarea que utilice registros SIMD, y el correspondiente soporte mediante la Excepción #NM y las modificaciones correspondientes en la función de cambio de contexto.

2. Modelo de programación SIMD

La práctica se divide en subsecciones, para cada una se sugiere un conjunto de instrucciones útiles para resolver los ejercicios.

2.1. Instr. de movimiento de datos y aritméticas básicas

Escribir las siguientes funciones en lenguaje ensamblador:

- *void SumarVectores(char *vectorA, char *vectorB, char *vectorResultado, int dimension)*
- *void RestarVectores(char *vectorA, char *vectorB, char *vectorResultado, int dimension)*

Resolver con las siguientes instrucciones:
Mov. de datos: ***movd***, ***modq***, ***movdqu***
Aritméticas: ***paddb***, ***paddw***, ***paddd***, ***paddq***, ***psubb***, ***psubw***, ***psubd***, ***psubq***

Además, se pide responder estas preguntas:

a) Qué sucede si la suma de dos componentes de los vectores supera el valor 255? Qué diferencia hay entre las instrucciones ***paddusw*** y ***paddsw***? Y entre ***paddusb*** y ***paddsb***?

b) Que cambios debería hacerles a las funciones anteriores si el tipo de datos es:
a) *short*, b) *int* y c) *long long int*?

Nota: Puede asumir que la dimensión de los vectores es de un tamaño múltiplo de la cantidad de elementos que procesa simultáneamente.

2.2. Instr. de comparación, lógicas y de nivel de bit

Escriba las siguientes funciones en lenguaje ensamblador:

- *void InicializarVectorEnCero(char *vectorA, int dimension)*
- *void InicializarVector(short *vectorA, short valorInicial, int dimension)*
- *void MultiplicarVectorPorPontenciaDeDos(int *vectorA, int potencia, int dimension)*
- *void DividirVectorPorPontenciaDeDos(int *vectorA, int potencia, int dimension)*
- *void FiltrarMayoresA(short *vectorA, short umbral, int dimension)*
  Pone en **unos** (*0xF..F*) aquellos elementos del vector cuyo valor es mayor a *umbral* y en **ceros** (*0x0..0*) aquellos elementos que son **menores iguales**.

Resolver con las siguientes instrucciones:
Comparación: ***pcmpgtb***, ***pcmpgtw***, ***pcmpgtd***, ***pcmpeqb***, ***pcmpeqw***, ***pcmpeqd***
Lógicas: ***pand***, ***por***, ***pxor***, ***pandn***
Nivel de Bit: ***psrlw***, ***psrld psrlq***, ***psrldq***, ***pslld***,***psllq***,***pslldq***

a) Qué cambios debería hacerles a las funciones anteriores en caso de que la dimensión de los vectores no sea múltiplo de la cantidad de elementos que procesa simultáneamente?

2.3. Instr. de movimiento de datos y aritméticas complejas

Escribir las siguientes funciones en lenguaje ensamblador:

- *void ExtenderTamañoVector(unsigned char *vector, unsigned short *vectorExtendido, int dimension)*
  Para cada elemento de *vector*, cuyo tamaño es de **1 Byte**, lo extiende a **1 Word** preservando su signo **positivo**.
- *void MultiplicarVectores(short *vectorA, short *vectorB, int *vectorResultado, int dimension)*
- *int ProductoInterno(short *vectorA, short *vectorB, int dimension)*
- *void Maximos(char *vectorA, char *vectorB, char *vectorResultado, int dimension)*
- *void SepararMaximosYMinimos(char *vectorA, char *vectorB, int dimension)*
  Deja en *vectorA* los máximos y en *vectorB* los mínimos. Es decir, para cada i, *vectorA[i]* $= max($*vectorA[i]*,*vectorB[i]*$)$ y *vectorB[i]* $= min($*vectorA[i]*,*vectorB[i]*$)$
- *void SumarYRestarVectores(int *vectorA, int *vectorB, int* vectorResultado, int dimension)*
  Es decir, el *vectorResultado* tiene que seguir el siguiente patrón:

$$vectorResultado = (a_1 + b_1, a_2 - b_2, a_3 + b_3, a_4 - b_4, ...)$$

Resolver con las siguientes instrucciones:
Desempaquetado: ***punpcklbw***, ***punpcklwd***, ***punpckldq***, ***punpckhbw***, ***punpckhwd***, ***punpckhdq***
Aritméticas: ***pmullw***, ***pmulld***, ***pmulhw***, ***pmulhd***, ***pmaddwd***, ***pmaxub***, ***pmaxuw***, ***pmaxud***, ***pminub***, ***pminuw***, ***pminud***
Empaquetado: ***packsswb packssdw***, ***packuswb packusdw***

a) Qué cambios habría que hacerle a la función **ExtenderTamañoVector** para que pueda extender elementos con signos?

b) Qué cambios habría que hacerle a la función **MultiplicarVectores** si el tipo de datos de los elementos de *vectorResultado* fuese *short*?

c) Qué diferencia hay entre las instrucciones ***packuswb*** y ***packsswb***?

2.4. Instr. de reordenamiento de datos

Escriba las siguientes funciones en lenguaje ensamblador:

- *void Intercalar(char *vectorA, char *vectorB, char *vectorResultado, int dimension)*

Resolver con las siguientes instrucciones:reordenamiento: ***pshufb***, ***pshufw***, ***pshufd***

a) Cómo haría función *Intercalar* si no dispone de las instrucciones de reordenamiento? Dé 2 maneras alternativas.

2.5. Instr. de punto flotante

Escriba las siguientes funciones en lenguaje ensamblador:

- *void SumarVector(float *vectorA, float *vectorB, float *vectorResultado, int dimension)*
- *void RestarVector(float *vectorA, float *vectorB, float *vectorResultado, int dimension)*
- *void MultiplicarVector(float *vectorA, float *vectorB, float *vectorResultado, int dimension)*
- *void DividirVector(float *vectorA, float *vectorB, float *vectorResultado, int dimension)*
- *void NormalizarVector(float *vectorA, float *vectorResultado, int dimension)*

Resolver con las siguientes instrucciones:
Mov. de datos: ***movups***, ***movaps***, ***movupd***, ***movapd***
Aritméticas: ***addps***, ***addpd***, ***subps***, ***subpd***, ***mulps***, ***mulpd***, ***divps***, ***divpd***, ***sqrtps***, ***sqrtpd***

a) Qué cambios debería hacerles a las funciones anteriores si el tipo de dato ahora es punto flotante de **doble precisión**?

b) Qué diferencia hay entre la instrucción ***addps*** y ***addss***? Y entre ***addpd*** y ***addsd***?

2.6. Instr. de conversión de tipos de datos

Escriba las siguientes funciones en lenguaje ensamblador:

- *void ProductoEscalar(short *vectorA, float escalar, int dimension)*
- *void ParteEntera(float *vectorA, int *vectorResultado, int dimension)*
- *void Normalizar(int *vectorA, float *vectorNormalizado, int dimension)*

Resolver con las siguientes instrucciones:conversión: ***cvtdq2ps***, ***cvtps2dq***, ***cvtdq2pd***, ***cvtpd2dq***

a) Qué cambios debería hacerles a las funciones anteriores si el tipo de dato ahora es punto flotante de **doble precisión**?

2.7. Ejercicios sobre imágenes

Una imagen está representada por una matriz de *alto* $\times$ *ancho*, donde cada posición representa un punto de color de la misma. A este punto se lo denomina *pixel*. En una imagen en escala de grises (de 8 bits), los colores varían desde el 0 (color negro) al 255 (color blanco) y cada *pixel*, desde luego, ocupa **1 byte**.

A su vez, en una imagen a color (de 24 bits), cada pixel está formado por 3 componentes (o canales): el rojo (**R**), el verde (**G**) y el azul (**B**). Cada una de estas componentes ocupa **1 Byte** (haciendo que el tamaño total del pixel sea de **3 Bytes**) y las distintas combinaciones posibles de las 3 generan los diferentes colores (la disposición en memoria de estás componentes es: primera la azul, luego la verde y por última la roja).

2.7.1. Inversión

Se desea realizar una función que dada una imagen en escala de grises invierta los colores, es decir, el realice el cálculo $255 - p$ para todos los pixeles p de la imagen de origen.

El prototipo de la función es:
*void Invertir(unsigned char *imagenFuente, unsigned char *imagenDestino, int alto, int ancho)*

2.7.2. Conversión color a escala de grises

Se desea realizar una función que dada una imagen en color retorne genere la misma imagen pero en escala de grises. Para hacer esto, se emplea la siguiente función:

$$imagenDestino(p) = max(R, G, B) \text{ para todo pixel } p \text{ de la } imagenFuente$$

El prototipo de la función es:
*void MonocromatizarInfinito(unsigned char *imagenFuente, unsigned char *imagenDestino, int alto, int ancho)*

2.7.3. Combinación

Se desea implementar la función combinar que dadas 2 imágenes de igual tamaño y en escala de grises retorna una tercera formada a partir de estas 2. Cada pixel de la imagen generada se forma de la siguiente manera:

$$imagenDestino(i,j) = \frac{alpha \cdot (imagenFuenteA(i,j) - imagenFuenteB(i,j))}{255} + imgB(i,j)$$

El prototipo de la función es:
void Combinar (unsigned char imagenFuenteA, unsigned char* imagenFuenteB, unsigned char* imagenDestino, int ancho, int alto, float alpha)*

2.7.4. Separación de componentes de color

Se desea realizar una función que dada una imagen en color retorne 3 imágenes en escala de grises, donde la primera imagen está formada por las componentes rojas (R) de la imagen de entrada, la segunda por las componentes verdes (G) y la última por las componentes azules (B).

El prototipo de la función es:
*void SepararComponentes(unsigned char *imagenFuente, unsigned char *imagenDestinoR, unsigned char *imagenDestinoG, unsigned char *imagenDestinoB, int alto, int ancho)*

2.7.5. Umbralizar

Implementar la función umbralizar que genera una imagen de tres colores, blanco, gris y negro determinada por la imagen fuente respetando la siguiente función:

$$imagenDestino(p) = \begin{cases} 0 & p \leq umbral\_minimo \\ 128 & umbral\_minimo < p \leq umbral\_maximo \\ 255 & p > umbral\_maximo \end{cases}$$

para todo pixel p de $imagenFuente$

El prototipo de la función es:
void Umbralizar (unsigned char imagenFuente, unsigned char* imagenDestino, int ancho, int alto)*

2.7.6. Transformaciones Morfológicas - Erosion

Implementar la función de erosión que le aplica la siguiente operación a la imagen (en escala de grises) de entrada:

$$imgD(i,j) = min(\begin{array}{ccccc} imgF(i-1,j-1) & , & imgF(i-1,j) & , & imgF(i-1,j+1) \\ imgF(i+1,j-1) & , & imgF(i+1,j) & , & imgF(i-1,j+1) \\ imgF(i,j=1) & , & imgF(i,j) & , & imgF(i,j+1)) \end{array}$$

Donde $imgF$ es la imagen fuente y $imgD$ el destino. Los indices i y j corresponden a las coordenadas en la imagen.

El prototipo de la función es:
*void Erosion(unsigned char imagenFuente, unsigned char *imagenDestino, int alto, int ancho)*

Determinar el rango en el cual se puede realizar esta cuenta, para evitar indefiniciones. Fuera del rango, la matriz resultante no debe ser modificada.

2.7.7. Suavizado

Implementar la función de suavizado que le aplica la siguiente operación a la imagen (en escala de grises) de entrada:

$$\begin{array}{rccccccccc} imgD(i,j) = & imgF(i-1,j-1)\cdot 1/16 & + & imgF(i-1,j)\cdot 2/16 & + & imgF(i-1,j+1)\cdot 1/16 & + \\ & imgF(i+0,j-1)\cdot 2/16 & + & imgF(i+0,j)\cdot 4/16 & + & imgF(i+0,j+1)\cdot 2/16 & + \\ & imgF(i+1,j-1)\cdot 1/16 & + & imgF(i+1,j)\cdot 2/16 & + & imgF(i+1,j+1)\cdot 1/16 & \end{array}$$

Donde $imgF$ es la imagen fuente y $imgD$ el destino. Los indices i y j corresponden a las coordenadas en la imagen.

El prototipo de la función es:
*void Suavizar(unsigned char imagenFuente, unsigned char *imagenDestino, int alto, int ancho)*

Determinar el rango en el cual se puede realizar esta cuenta, para evitar indefiniciones. Fuera del rango, la matriz resultante no debe ser modificada.

Nota: Para generar el pixel $imagenDestino(i,j)$ en la imagen destino, se debe operar sobre enteros, por lo que la aplicación de la función descripta debe ser adaptada para tal caso.

2.7.8. Promediar vecinos

Implementar la función promediar vecinos que dada una imagen (en escala de grises) de entrada, le aplica la siguiente fórmula a todos los píxeles:

$$imgD(i,j) = \frac{imgF(i,j-1)+imgF(i,j+1)+imgF(i-1,j)+imgF(i+1,j)}{4}$$

Donde $imgF$ es la imagen fuente y $imgD$ el destino. Los indices i y j corresponden a las coordenadas en la imagen.

El prototipo de la función es:
void PromediarVecinos(unsigned char imagenFuente, unsigned char* imagenDestino, int alto, int ancho)*

Determinar el rango en el cual se puede realizar esta cuenta, para evitar indefiniciones. Fuera del rango, la matriz resultante no debe ser modificada.