Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Ciencias de la Computación

Organización del computador II
Segundo cuatrimestre 2009

Grupo: POPA

Apellido y nombre	L.U.	mail
Cerrutti, Mariano Javier	525/07	vscorza@gmail.com
Huel, Federico Ariel	329/07	federico.huel@gmail.com
Mita, Rogelio Iván	635/07	rogeliomita@gmail.com

5 de Octubre de 2009

Índice

1. Archivos adjuntos

Implementación

- src:
  - *bordes.c*
  - *asmSobel.asm*
  - *asmPrewitt.asm*
  - *asmRoberts.asm*
- img:
  - *lena.bmp*

Includes

- *offset.inc*
- *macros.mac*

Enunciado

- *EnunciadoTP1A.pdf*

Informe

- *tp1-a.pdf*

2. Instrucciones de uso

Decidimos escribir los códigos de las funciones de forma separada para mayor claridad. Utilizamos también un archivo (*macros.mac*) para definir nuestras macros y (*offset.inc*) para darle nombres declarativos a los datos de la estructura imagen de opencv. En el cd adjunto al trabajo práctico, se encuentran todos los archivos fuente clasificados según el tipo. Decidimos utilizar un makefile para compilar todos los archivos a la vez.

3. Introducción

Este trabajo consistió en escribir en lenguaje ensamblador distintas funciones para buscar bordes de imagen, Tanto en función de x, como en y, o ambas. La razón de escribir código en lenguaje de bajo nivel radica en que es imperativo conocer el manejo interno más básico de las instrucciones de la arquitectura del computador.

Las funciones en código assembler estan divididas según el operador que se desea aplicar a la imagen para buscar bordes.

4. Desarrollo

Antes de mostrar los códigos de cada función explicamos algunas cuestiones generales a todos los algoritmos:

- Nos fue pedido que en todos los algoritmos sea respetada la convención C, de forma que los mismos pasos (guardado de los registros edi, esi y ebx, ajuste de la pila, etc) se encuentran al principio de cada función.
- En todas las funciones, consideramos que los parámetros (excepto los que se pedían) no debían ser modificados, y por lo tanto, los mismos los guardamos en variables locales o bien registros. En algunos casos esto es muy importante, por ejemplo, cuando nos pasan un puntero a una lista, si modificamos ese puntero, se pierde la dirección de esa lista luego de llamar a la función, y esto no puede ocurrir.
- También tuvimos en cuenta que las imágenes se guardan en memoria alineadas
- Por cuestiones de simpleza decidimos en los algoritmos también procesar la basura de la imagen ya que la libreria opencv cuando hace el resize de la imagen se limita al ancho y alto dado como parametro, y ademas esta decición no cambia en absoluto la visualización de la imagen resultante.

A continuación se exponen las funciones implementadas, junto con sus respectivos códigos en assembler y se explica a su vez los registros especiales utilizados y su funcionamiento en líneas generales.

4.1. offset.inc

Aqui estan los offset a datos que usamos de la estructura imagen del openCv, para mayor claridad

```
1  ; typedef struct _IplImage
2  ; {
3  ;       int nSize;
4  ;       int ID;
5  ;       int nChannels;
6  ;       int alphaChannel;
7  ;       int depth;
8  ;       char colorModel[4];
9  ;       char channelSeq[4];
10 ;       int dataOrder;
11 ;       int origin;
```

```
;       int align;
;       int width;
;       int height;
;       struct _IplROI *roi;
;       struct _IplImage *maskROI;
;       void *imageId;
;       struct _IplTileInfo *tileInfo;
;       int imageSize;
;       char *imageData;
;       int widthStep;
;       int BorderMode[4];
;       int BorderConst[4];
;       char *imageDataOrigin;
; }
; IplImage;

%define DEPTH           16
%define WIDTH           40
%define HEIGHT          44
%define IMAGE_DATA      68
%define WIDTH_STEP      72
```

4.2. macros.mac

Estos son las macros utilizadas en cada funcion.

```
;=============================
; MACRO doEnter
;=============================
; Escribe el encabezado que arma el stack frame
; Entrada:
;       tamanio         tamanio de memoria a reservar al entrar al proc
;=============================
%macro doEnter 0-1 0
        push ebp
        mov ebp, esp
%if %1 <> 0
        sub esp, %1
%endif
        push edi
        push esi
        push ebx
%endmacro
;=============================
; MACRO doLeave
;=============================
; Escribe la salida que restaura el stack frame previo
; Entrada:
;       tamanio         tamanio de memoria reservada al entrar al proc
;       doRet           se marca en 1 si debe llamar a ret
;=============================
%macro doLeave 0-2 0,0
```

```
27          pop ebx
28          pop esi
29          pop edi
30  %if %1 <> 0
31          add esp, %1
32  %endif
33          pop ebp
34  %if %2 <> 0
35          ret
36  %endif
37  %endmacro
38
39  ;=========================
40  ; MACRO doWrite
41  ;=========================
42  ; Escribe una cadena a consola
43  ; Entrada:
44  ;       mensaje         direccion de comienzo de la cadena
45  ;       len             largo de la cadena a escribir
46  ;=========================
47  %macro doWrite 1
48          %%msg: db %1
49          %%len: equ $- %%msg
50          mov eax,4                       ;inicializa escritura a consola
51          mov ebx,1
52          mov ecx,%%msg
53          mov edx,%%len
54          int 80h
55  %endmacro
56  ;=========================
57  ; MACRO doEnd
58  ;=========================
59  ; Termina la ejecucion con el codigo deseado
60  ; Entrada:
61  ;       codigo          codigo de error deseado, cero en su defecto
62  ;=========================
63  %macro doEnd 0-1 0
64          mov eax,1
65          mov ebx,%1
66          int 80h
67  %endmacro
68  ;=========================
69  ; MACRO doMalloc
70  ;=========================
71  ; Pide la cantidad especificada de memoria
72  ; Entrada:
73  ;       cantidad        cantidad de memoria a reserver
74  ;=========================
75  %macro doMalloc 1
76          push %1
77          call malloc
78          add esp, 4
79  %endmacro
```

```
80  ;=============================
81  ; MACRO doRetc
82  ;=============================
83  ; Retorna si se cumple la condicion especificada
84  ; Entrada:
85  ;       condicion       condicion ante la cual retornar
86  ;=============================
87  %macro  doRetc 1
88          j%-1    %%skip
89          ret
90    %%skip:
91  %endmacro
92  ;=============================
93  ; MACRO doWriteFile
94  ;=============================
95  ; Escribe a archivo solo si esta definido el DEBUG
96  ; Entrada:
97  ;       arch            puntero al archivo
98  ;       msg             texto a escribir
99  ; Uso:
100 ;       doWriteFile [FHND], {"hola mundo",13,10}
101 ;=============================
102 %macro  doWriteFile 2+
103 %ifdef DEBUG
104         jmp     %%endstr
105   %%str:        db      %2
106   %%endstr:
107         mov     dx,%%str
108         mov     cx,%%endstr-%%str
109         mov     bx,%1
110         mov     ah,0x40
111         int     0x21
112 %endif
113 %endmacro
```

4.3. Recorrido de la matriz de imagen

1. **Cosas que tuvimos en cuenta**

   La matriz la recorrimos procesando la basura

   Cada pixel ocupa exactamente un byte, ya que la imagen que recive la función es en escala de grises

   En la arquitectura IA-32 los datos en memoria son guardados de forma *little-endian*, de forma que el byte más significativo de un dato se encuetra en la posición de memoria más alta.

4.4. Funciones implementadas

- Intentamos reducir el acceso a memoria utilizando registros para utilizar en aquellas operaciones que se encontraban dentro de los ciclos, para evitar multiples accesos a memoria. Si bien esto llevó a tener un codigo menos entendible a simple vista, lo explicaremos en detalle.

- Variables locales

  SRC Contiene el puntero a la imagen de entrada.

  DST Contiene el puntero a la imagen resultado.

  WIDTH Contiene el tamano de ancho de la imagen.

  HEIGHT Contiene la altura de la imagen.

  XORDER Variable que especifica si el calculo debe hacerse sobre la derivada X.

  YORDER Variable que especifica si el calculo debe hacerse sobre la derivada Y.

  T_HEIGHT Variable que se usa como contador para el recorrido vertical.

- Registros de propósito general

  eax Se utiliza como acumulador, para realizar los calculos de los pixeles.

  ebx Se utiliza como acumulador, para realizar los calculos de los pixeles.

  ecx Se utiliza para recorrer la imagen de entrada.

  edx Se utiliza como contador para el recorrido horizontal de la imagen.

  esi Se utiliza para recorrer la imagen de salida.

  edi Contiene el ancho de la imagen.

- **Funcionamiento**

  Lo primero que hacen todos los algoritmos es ubicar al registro ecx en el comienzo de la imagen de entrada y a esi en el primer pixel que se modificará en la imagen de salida. Luego comienza el ciclo que recorre la matriz verticalmente, para lo que carga en edx el ancho de la imagen,para usarlo como contador del recorrido horizontal, sacándole los pixeles laterales ya que no se procesarán debido al cálculo.

4.4.1. Sobel

1. **Cosas que tuvimos en cuenta**

   Lo que tuvimos en cuenta si tuvimos algo.

2. **X Sobel**

   Dentro del ciclo que recorre la imagen horizontalmente, se copian en eax los cuatro pixeles que se encuentran a continuación en la imagen y mediante una máscara se obtienen sólo el primero y el tercero. Luego se hace lo mismo para los pixeles que se encuentran en la siguiente línea guardándolos en ebx, y shifteándolos un bit a izquierda para duplicar su valor. Se suman ambos registros y y se gurada el resultado en eax. Se cargan los pixeles de la tercer línea en ebx, se le aplica una máscara para obtener sólo los necesarios y se los suma a los acumulados quedando en eax la parte alta de eax la suman de los primeros pixeles de cada fila y en la parte alta los terceros. Luego, se copia eax a ebx, y se lo shiftea (a eax)para que queden los pixeles que deben restarse en la parte menos significativa. Con los datos listos, se realiza la resta.

3. **Y Sobel**

   Dentro del ciclo que recorre la imagen horizontalmente, se cargan en eax y en ebx 4 pixeles en los que los primeros 3 contienen en orden los pixeles que serán sumados de acuerdo al cálculo de Sobel. Luego se aplica una máscara a eax que deja solo el primer y tercer pixel y otra en ebx que deja solo el segundo. Se hace un shift a derecha a ebx para ubicar el pixel en la parte baja del registro para realizar la suma con los otros pixeles, pero dejando un bit a derecha para duplicar su valor. A continuación se suman el primer y segundo(duplicado en valor) pixel, se shiftea el tercero y se lo suma a los dos anteriores, quedando en el registro ebx el valor de la suma de los pixeles. Luego se copian en eax los cuatro pixeles que contienen a los que deben restarse y se elimina el que no se necesita mediante la utilización de una máscara. Para poder operar, se utiliza la función ror para dejar en la word menos significativa solo un pixel y poder restarlo llamando a ax. Luego se borra la parte baja de eax y se guarda la parte alta en ebx (La parte alta de ebx no estaba siendo utilizada ya que acumula solo la suma de tres Words). Se shiftea eax para que quede el tercer pixel en la parte baja y se lo resta al acumulador. Lo mismo se hace con el segundo pixel, pero dejando un bit a derecha al shiftearlo para que su valor sea el doble.

4. **Ambas derivadas**

   Antes de copiarlo, se comprueba que el pixel calculado no se haya saturado, y en caso de que eso haya sucedido lo que se copia es el mayor o menor valor para el pixel, de acuerdo a la saturacion, evitando desbordes. Luego se restan los contadores corresopondientes y se comprueba si ha finalizado alguno de los ciclos para decidir que salto se raliza.

5. **Código**

```
1 ACA VA EL CODIGO
```

4.4.2. Prewitt

1. **Cosas que tuvimos en cuenta**

   Lo que tuvimos en cuenta si tuvimos algo.

2. **X Prewitt**

   Dentro del ciclo horizontal, se copian en eax los cuatro pixeles que se encuentran a continuación en la imagen y mediante una máscara se obtienen solo el primero y el tercero. Luego se hace lo mismo para los pixeles que se encuentran en la siguiente linea guardándolos en ebx. Se suman ambos registros y y se gurada el resultado en eax. Se cargan los pixeles de la tercer línea en ebx, se le aplica una máscara para obtener solo los necesarios y se los suma a los acumulados quedando en eax la parte alta de eax la suman de los primeros pixeles de cada fila y en la parte alta los terceros. Luego, se copia eax a ebx, y se lo shiftea 16 bits a derecha(a eax)para que queden los pixeles que deben restarse en la parte menos significativa. Con los datos listos, se realiza la resta.

3. **Y Prewitt**

   Dentro del ciclo que recorre la imagen horizontalmente, se cargan en eax y en ebx 4 pixeles en los que los primeros 3 contienen en orden los pixeles que serán sumados de acuerdo al cálculo de Sobel. Luego se aplica una máscara a eax que deja solo el primer y tercer pixel y otra en ebx que deja solo el segundo. Se hace un shift a derecha a ebx para ubicar el pixel en la parte baja del registro para realizar la suma con los otros pixeles. A continuación se suman el primer y segundo pixel, se shiftea el tercero y se lo suma a los dos anteriores, quedando en el registro ebx el valor de la suma de los pixeles. Luego se copian en eax los cuatro pixeles que contienen a los que deben restarse y se elimina el que no se necesita mediante la utilización de una máscara. Para poder operar, se utiliza la función ror para dejar en la word menos significativa solo un pixel y poder restarlo llamando a ax. Luego se borra la parte baja de eax y se guarda la parte alta en ebx (La parte alta de ebx no estaba siendo utilizada ya que acumula solo la suma de tres Words). Se shiftea eax para que quede el tercer pixel en la parte baja y se lo resta al acumulador. Lo mismo se hace con el segundo pixel, pero dejando un bit a derecha al shiftearlo para que su valor sea el doble.

4. **Ambas derivadas**

   Antes de copiarlo, se comprueba que el pixel calculado no se haya saturado, y en caso de que eso haya sucedido lo que se copia es el mayor o menor valor para el pixel, de acuerdo a la saturacion, evitando desbordes. Luego se restan los contadores corresopondientes y se comprueba si ha finalizado alguno de los ciclos para decidir que salto se raliza.

5. **Código**

```
1 ACA VA EL CODIGO
```

4.4.3. Roberts

1. **Cosas que tuvimos en cuenta**

   Lo que tuvimos en cuenta si tuvimos algo.

2. **X Roberts**

   Comienza el ciclo horizontal. Se cargan los tres pixeles a los que está apuntando ecx (que es el que recorre la imagen) y se utiliza una máscara para quedarse solo con el primer pixel en ax, ya que es el que se utilizará en el cálculo. Se copian en bx los dos pixeles que se encuentran en la siguiente linea, se lo enmascara para quedarse con el más significativo y se lo shiftea para poder operar con el pixel guardado en ax. Luego se realiza la resta entre bx y ax, se verifica si hay saturación y se copia el pixel guardado en al en la imagen de salida.

3. **Y Roberts**

   En el ciclo horizontal, se copian en bx los pixeles necesarios quedándose solo con el segundo, que es el que se utilizará. Se lo shiftea 8 bits a derecha para que quede en la parte menos significativa y que se pueda operar con otro pixel. Luego se copia el pixel que está en la linea siguiente en ax usando una máscara para filtrar el byte más significativo y se realiza la resta entre bx y ax. Se verifica la aparición de saturación y se copia el pixel en lugar correspondiente de la imagen de salida. Se incrementa los registros que recorren las imágenes y se aumenta de línea si no deben escribirse más pixeles en dicha línea y vuelve a comenzar el ciclo.

4. **Ambas derivadas**

   Antes de copiarlo, se comprueba que el pixel calculado no se haya saturado, y en caso de que eso haya sucedido lo que se copia es el mayor o menor valor para el pixel, de acuerdo a la saturacion, evitando desbordes. Luego se restan los contadores corresopondientes y se comprueba si ha finalizado alguno de los ciclos para decidir que salto se raliza.

5. **Código**

```
1 ACA VA EL CODIGO
```

5. Resultados

COMPLETAR CON RESULTADOS

6. Conclusiones

Luego de implementar este trabajo en assembler, podemos concluir que si bien nos da una gran libertad a la hora de programar (modificar directamente la memoria, trabajar con los registros directamente, etc.), la claridad del código de estos programas no es tan claro como los de lenguaje de alto nivel que manejamos *C*, *C++*, etc. Vemos esto como una desventaja ya que teniendo en cuenta que el trabajo es grupal, en ocasiones tuvimos que repartir las tareas, es decir, algunas funciones, y luego, al juntarnos y ver las implementaciones que había hecho cada uno se dificultaba entenderlas.
De todas formas, consideramos que la realización de este trabajo nos sirvió para obtener un importante conocimiento sobre el lenguaje de bajo nivel, el cual complementa entender cosas de lenguajes de más alto nivel, y consideramos además que es una muy buena práctica para ayudarnos a comprender el funcionamiento interno de un computador.