BLM 2021 Alt Seviye Programlama

2022 - 2023 Güz Dönemi 2. Ödev

Gray Resimde Morfolojik işlemler

Ders Yürütücüsü: FURKAN ÇAKMAK

Ödevi Yapan: Berkay Ateş

No: 21011609

berkay.ates1@std.yildiz.edu.tr

15.01.2023

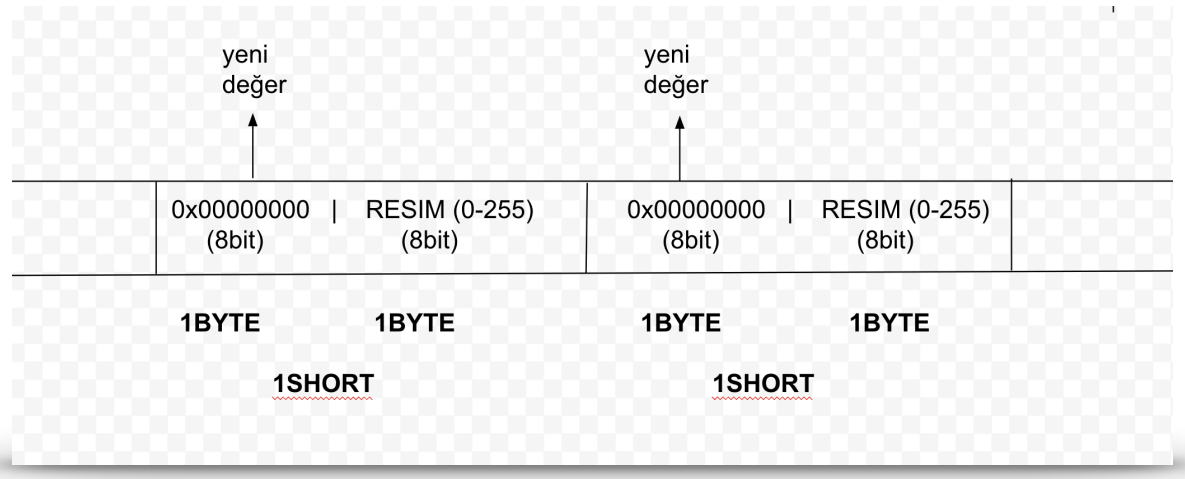
İçindekiler

Genel Açıklama

Dilation veya Erosion işlemlerini resim matrisi üzerinde uygularken herhangi bir şekilde orijinal datanın korunması ve yapılan morfolojik işlem sonucu çıkan yeni resmin da farklı bir yerde saklanması gerekmektedir. Bunun için görünürde **2 farklı** yol vardı.

1. Yol (TERCİH EDİLMEYEN)

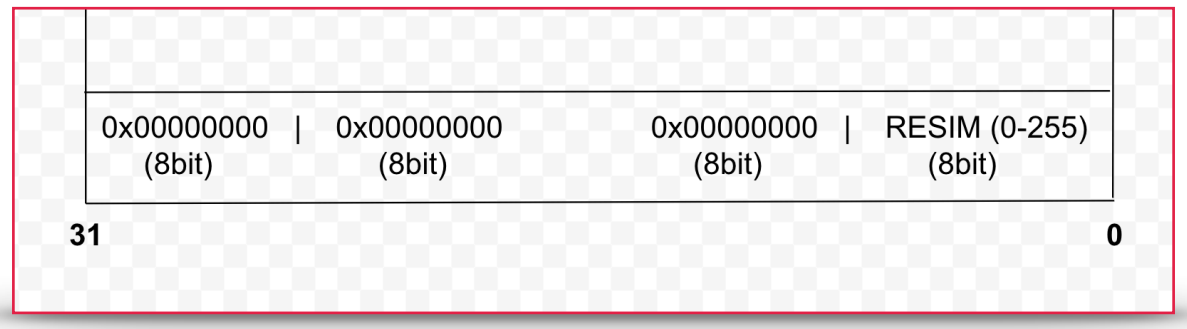
Resim gray scale olduğu icin pixellerin değerleri 0-255 aralığında değişiyor dolayısıyla resmin değerlerini 8 bite veya da 1 byte büyüklüğünde bir alana sığdırılabilir. Fakat **CPP (c++)** içerisinde kullanabileceğimiz ve intager türünde veri depolayabileceğimiz en küçük alan SHORT tipinde ve short verilerde 2 byte büyükliğünde. Dolayısıyla resim CPP içerisinde bulunduğu short büyüklüğündeki alanın her daim düşük anlamlı 8 bitinde bulunuyor. Short değerin yüksek anlamlı 8 bit değeri ise her daim sıfırlarla dolu. Bu durumdan istifade ederek Short verilerin üst anlamlı 8 bitinde morfolojik işlem sonucu oluşan resmi saklayarak ve ROL,SHR gibi bit bazındaki işlemlerle morfolojik işlemlerimizi gerçekleyebiliriz. Assembly ile çalıştığımızı farkederek ve sahip olduğumuz alanı verimli kullanarak fazladan bir depolama alanına ihtiyaç duymuyoruz. Aşağıda bahsedilmek istenilen resmedilmiştir.



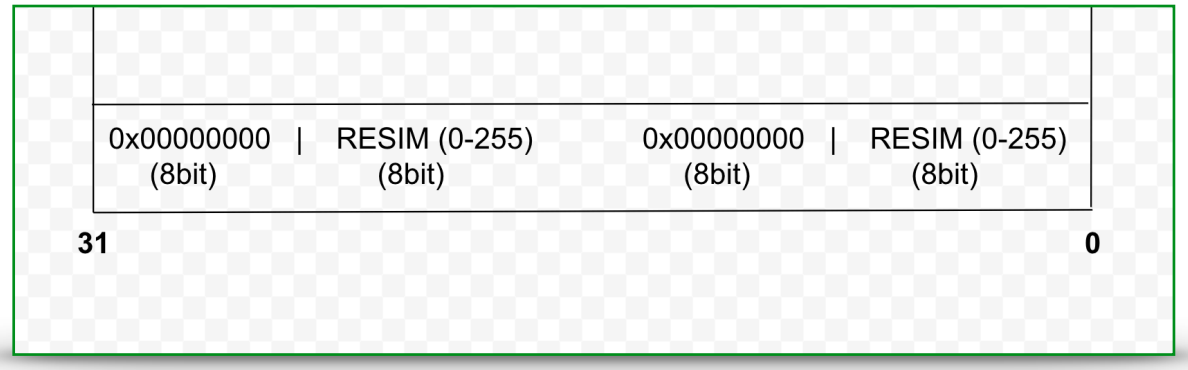
2. Yol (UYGULANMIŞ OLAN)

İlk yol sonradan farkedildi için uygulanmamış olup 2. Yol uygulanmıştır. 2. Yöntemde ise dizinin tamamını stack üzerine gönderdim ve orijinal resim üzerinde filtreyi gezdirdim. İşlem sonucu stack üzerindeki resme yeni verileri yazdım. Bu işlem sonucunda Stackte yeni resim oluşmuş oldu. Morfolojik işlem bittikten sonra da Stack üzerindeki veriyi asıl resmimiz üzerine yazarak işlemi sonlandırdım.

Bu yöntemi uygularken her bir pixel değeri resim dizisinden word(2byte) olarak cast edip sonrasında stacke attığımda Stackte aşağıdaki gibi bir durum oluştu ve STACKOVERFLOW hatası ile karşılaştım.



Sonrasında resmin pixel değerlerini Stack üzerine tek tek atmaktansa ikili ikili atabileceğimi farkettim ve resim dizisinden word yerine **dword** olarak cast yaparak Stack yapısını doldurdum ve herhangi bir Stackoverflow hatası ile karşılaşmadım. Dword cast ettiğimde ise Stack aşağıdaki gibi bir durum aldı. Dword cast ederek her bir pixelin Stack üzerindeki maliyetini 32 bitten 16 bite düşürdüğümüz için Stack taşmadı.



ADIM ADIM Çözüm

Dilation

1.Adım Stack üzerine yollamak

Bu işlemde resmi stack üzerine resmin birinci pixeli en tepede kalacak şekilde push yaparak yolluyoruz.

```
67          // resmi asagidan yukari dogru stacke atalim
68          mov eax,n
69          shl eax,1
70          add eax,resim_org
71          mov edi,eax             // edi resmin en sonunda,casting
72          sub edi,4               // işlemi yüksek adrese dogru veri alacagi
73          mov ecx,n               // icin edi'yi azaltmamiz lazim
74          shr ecx,1
75  cpstc:  mov eax, dword ptr [edi]
76          push eax
77          sub edi,4
78          loop cpstc
79
```

Mov eax,n

-> pixel sayısını eax' e alıyoruz

shl eax,1

-> pixeller word tanımlı oldukları için dizinin uzunluğunun 2 katı kadar byte elimizde var. Bu yüzden eax i 2 ile çarpmamız lazım.

add eax,resim_org

-> resim dizisinin başlangıç addrsini yani offsetini eax ile topluyoruz. Böylece resim dizisinin en sonuna gelmiş olduk.

mov edi,eax

-> adresi edi içerisine alarak edi sayesinde okuma yapabilelim

sub edi,4

-> casting işlemi yukarı doğru 4 byte getireceği için edi değerini 4 byte azaltıyoruz böylece dizinin dışına taşmadan resme ait olmayan verileri okumamış oluyoruz etmemiş oluyoruz.

mov ecx,n

shl ecx,1

-> dword cast ederek pixelleri 2, 2 stack üzerine atacağımız içen **cpstc** loopu n/2 kadar dönmek zorunda. Dolayısıyla ecx içerisine n yani pixel degerinin yarısını koyalım .

cpstc:

mov eax, dword ptr[edi]

push eax

sub edi,4

loop cpstc

-> her defasında resim dizisinden dword cast ederek bu değeri Stack üzerine yolluyoruz.

2. Adım Resmin Boyutunu Bulmak

Bu adımda resmin boyutunu resim kare olduğu için 1'den başlayarak tüm sayıların karelerini alıyoruz ve resmin boyutuna ulaşana kadar bu işlemi devam ettiriyoruz. **(WHILE DONGUSU)**

```
83          xor ecx, ecx
84  sqr:    inc ecx
85          mov eax, ecx
86          mul ecx
87          cmp eax, n
88          jne sqr
89          mov ebx,ecx          // resmin boyutu suanda ebx icerisinde
```

Her defasında ecx değerini 1 arttırıyoruz eax'e ecx değerini koyarak ecx ile eax değerini çarpıyoruz. Sonra eax ile resimdeki pixel sayısının eşit olup olmadığını kontrol ediyoruz. Eğer eşitlerse resmin boyutu ecx içerisinde demektir değilse hala resmin boyutunu bulamamışız demektir. İşlem sonunda da ebx içerisine ecx de bulunan değeri kopyalıyoruz.

3. Adım Morfolojik İşlemin Uygulanması

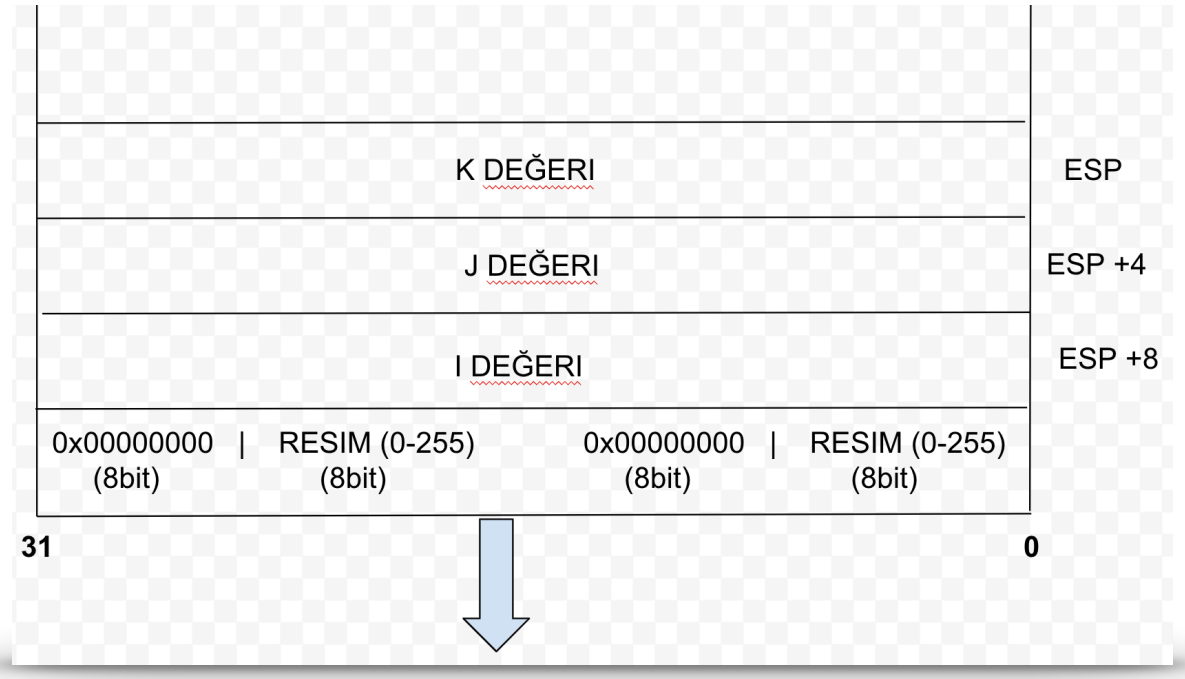
Assembly tarafında ecx değerlerimiz azalarak 0 olacağından resmin sağ alt köşesinden başlayarak tarama yapıyoruz. Tarama işlemini yaparken hem resim üzerinde hem de filtre üzerinde dolanmamız gerekiyor. Yani n^4 karmaşıklıkta bir işlemi gerçekleştirmemiz gerekiyor.

```
        mov ecx,ebx
        dec ecx                 //i ayarlandi
imgI:   push ecx

        mov ecx,ebx             // j ayarland˝
imgJ:   push ecx

        mov ecx,filter_size     // k ayarland˝
kerK:   push ecx

        mov ecx,filter_size     // L ayarlandi
kerL:   mov eax,filter_size
```

Yukarıdaki adımlarda oluşacak olan 4 for yapısının ecx değerlerini ayarlıyoruz.

kerL - KERNEL yani filtrenin sütunları

kerK - Kernelin yani filtrenin satırları

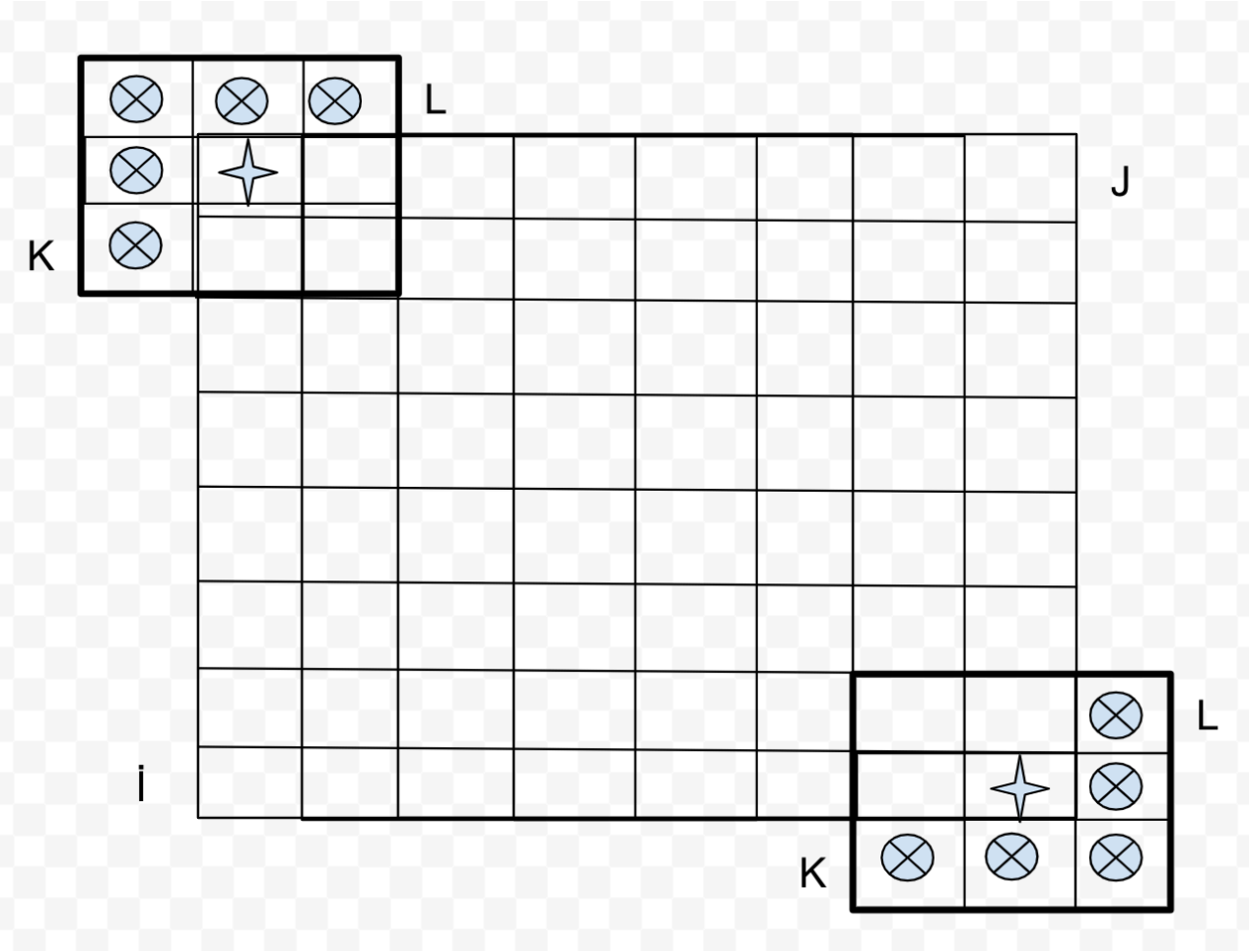


Döngü değişkenlerini ayarladığımızda stack yukarıdaki gibi bir hal alıyor. Stack uzerinden okuma yaparken **EBP** kullandığımda başka verilerin değerleride bozulduğu için stack üzerindeki işlemler ESP referans alınarak yapılmıştır. HICBIR DURUMDA ESP'NIN DEGERI **DEGISTIRILMEMISTIR.**

```
104
105 kerL:   mov eax,filter_size     // satir kontrolu
106         shr eax, 1
107         add eax, [esp + 8]
108         sub eax, [esp]          //eax icerisinde suanda i+filter_size/2-k degeri var
109         cmp eax, 0
110         jl ext
111         cmp eax, ebx
112         jnb ext
113         mov edx, filter_size    // sutun kontrolu
114         shr edx, 1
115         add edx, [esp + 4]
116         sub edx, ecx            // edx icerisinde suanda j+filter_size/2-L değeri var
117         cmp edx, 0
118         jl ext
119         cmp edx, ebx
120         jnb ext
121
```

Yukarıdaki işlemlerde ise en içteki forda o an kontrol edecek olduğumuz pixel noktasının resmin dışına taşıp taşmadığını bulmaya çalışıyoruz eğer taşmış işe **ext** labeline zıplayarak sonraki pixel değerine geçiyoruz. Kısacası o an bakmakta oluğumuz pixel aşağıdaki resimde yuvarlak çarpılara denk geliyorsa herhangi bir işlem yapmıyoruz. İçerdeki k-L loopu sonucunda da Yıldız işaretinin olduğu noktayı güncelleyerek işlemi bitiriyoruz.

YUKARIDAKİ kod çalıştığında **eax** içerisinde bakmak istediğimiz pixelin satır değeri **edx** içerisinde ise bakmak istediğimiz pixelin sütün değeri oluşuyor. Dolayısıyla eax ile resim boyutunu çarpıp edx ile toplarsak bakmak istediğimiz pixelin indisine erişmiş oluruz.



Sonrasında kontrol edeceğimiz indis dışarı taşmıyorsa bu pixelin adresini aşağıdaki kodda buluyoruz.

```
122         push edx
123         mul ebx
124         pop edx
125         add eax,edx             // resmin kacinci pixeli
126         shl eax,1               //kontrol edilecekse oradayiz suanda
127         add eax,resim_org
128         mov edi,eax             // edi icerisinde kernelin kontrol
129                                 // etmek istedigi noktanin addresi var suanda
130
```

Ebx içerisinde resmin boyutu vardı dolayısıyla eax(satır) ile ebx'i çarpıp edx(sütün) ile toplarsak pixelin indisine erişiriz. Fakat dizi word tanımlı olduğu için eax i 2 ile çarpmamız gerekir. Sadece 2 ile çarpmamız da yetmez birde resim_org ile toplamamız da gerekir ki hafızda doğru alana erişebilelim. İşlemin sonunda eax içerisinde bakmak istediğimiz pixel adresi oluşuyor. Bu pixel adresini edi içerisine alarak sonraki adımlarda bu pixele erişeceğiz.

EDİ -> FILTRENIN O AN BAKACAĞI PIXEL ADDRESINDA

```
131
132             jmp l1
133     l44:    jmp imgI
134
135             jmp l1
136     l33:    jmp imgJ
137
138             jmp l1
139     l22:    jmp kerK
140
141             jmp l1
142     l11:    jmp kerl
143
144     l1:     mov eax,[esp+8]
```

Tum bu kontroller icin yazmış oldugumuz kodlar LOOP komutunun zıplama eşiğini aştığı icin yukarıdaki gibi bir yapı kurgulanmıştır.

```
283 l1:     mov eax,[esp+8]
284         mul ebx
285         add eax,[esp+4]     // eax de resmin kernel merkezine gelen pixel numarasi var
286         test eax,eax
287         jc tek
288         shl eax,1
289         mov edx,[esp+eax+8] // dx'i kontrol edecegiz
290         mov ax,word ptr[edi]
291         cmp dx,ax
292         jb ext
293         mov dx,ax
294         jmp toStk
295 tek:    inc eax
```

Bir önceki adımda filtrenin kontrol edeceği alanı bulduk ve edi içerisine koyduk. Yukarıdaki resimde olduğu gibi edi'nin offsetini tuttuğu pixelle Kernelin tam orta noktasına gelen pixeli kıyaslamamız gerekiyor. Kernelin ortasına gelen alanı stack uzerinden okumamız gerek. Dolayısıyla resim pixellerini stack uzerine ikişer ikişer attığımız icin **[sp + i*img_size +j *2 + 8]** gibi bir yaklaşıma ihtiyacımız var.

+ 8 i,j,k degerlerinden dolayı

i*img_size +j *2 pixelleri stack uzerine ikişer ikişer attık diye

O an i veya j toplamının tek olmasını kontrol ederek de stackten gelen verinin üst 16 bitini veya alt 16 bitini işleyip işlemeyaceğimizi kontrol ediyoruz.

```
295  tek:    inc eax
296          shl eax,1
297          mov edx,[esp+eax+8]
298          mov ax,word ptr[edi]
299          push ecx
300          mov cl,16
301          rol edx,cl
302          pop ecx
303          cmp dx,ax
304          jb ext
305          mov dx,ax
306          push ecx
307          mov cl,16
308          rol edx,cl
309          pop ecx
```

Eğer i*img_size + j tek ise rol yaparak **edx'in** üst anlmlı kısmını alt anlamlı kısma düşürüyoruz. Böylelikle **dx, dh, dl** gibi registerlara işleyeceğimiz kernel merkezinde bulunan yere stack uzerinden erişebiliyoruz. Gerekli kıasyalamaları yaptıktan sonra (gerekirse guncelliyoruz)veriyi tekrar ROL komutu ile döndürerek eski şekline getiriyoruz.

Dilation icin orijinal veri kucukse güncelleme yapıyoruz.

Erorsion icin ise orijinal veri büyükse güncelleme yapıyoruz.

(TEK-CIFT DURUMUNU TEST KOMUTU ILE KONTROL EDIYORUZ)

```
310  toStk:  mov eax, [esp + 8]
311          push edx
312          mul ebx
313          pop edx
314          add eax, [esp + 4]
315          shl eax,1
316          mov [esp + eax + 8],edx
317
```

Kontrol ettiğimiz pixeli en sonunda stackten aldığımız noktaya yukarıdaki gibi yazarak tüm işlemlerimizi sonlandırıyoruz.

```
318  ext:    loop l11
319          pop ecx
320          loop l22
321          pop ecx
322          loop l33
323          pop ecx
324          loop l44
325
```

4.Adım Stacktan Verileri Diziye Geri Yazmak

```
330             mov ecx,n
331             shr ecx,1
332             mov edi,resim_org
333     stimg:  pop eax
334             mov dword ptr[edi],eax
335             add edi,4
336             loop stimg
```

Stack içerisine resmin ilk pixeli en yukarıda olacak sekilde push işlemi yapmıştık dolayısıyla resim_org addresinden başlayarak stackten sırayla gelencek olan 32 bitlik verileri dword cast ederek yazarsak morfolojik işlem uygulanmış olan resimi ilgili noktalara yazmış oluyoruz.

Özet

Değerleri stacke atıp sonrasında kernelin uzerinde gezdiği elemanlarla kıyaslayıp gerekirse güncelleyerek tekrar ilgili yere yazaıyoruz. Stackten veri gelirken kernel merkezindeki pixel indisi tek veya çift ise durumu ayarlamayı da unutmuyoruz. En sonunda verileri tekrar stackten diziye yazarak işlemi sonlandırıyoruz.

EROSION

Erosion işlemi ile dilation işlemi arasında “JA” komutunun “JB” olması haricinde bir değişiklik olmadığı için erosion kodu adım adım açıklanmamıştır.

#KAZANIMLAR

- Assembly ile çalışırken sahip oldugumuz bit bazındaki işlemlerin durumu nasıl kolaylaştırabileceğini tecrübeledim.
- Kullandığım TEST, ROL , SHR, SHL gibi komutlarla bit bazında işlemler yaptım. Rotate ve shift komutlarının ne işe yaradıkları konusunda iyice kafamda anlam uyanmış oldu.
- Kullandığım bilgisayarın macOS olması dolayısıyla Visual Studio yu kurmak icin windows sanal makinesi kurmak gibi bir yola girerek sanal makine nasıl çalışır windows işletim sisteminin arayüzü nasıldır gibi konularda tecrübe kazandım. (MAC de VS ile cpp derleyemediğim için bu yolu tercih ettim)
- Erosion ve Dilation gibi morfolojik image işlemleri konusunda fikir sahibi oldum.
- Inline assembly ile fonksiyonların kodlarını yazıp işlemleri hızlandırma konusunda tecrübe sahibi oldum
- Stack yapısının gerektiğinde verileri nasıl saklayabileceğimiz bir alan gorevi üstlenebilecegini gördüm.