



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Deney 5

Görüntü İşleme: C & Çevirici

1. GİRİŞ

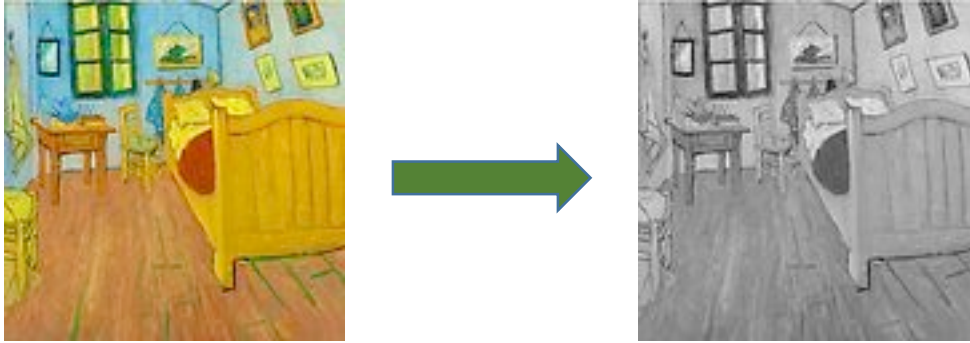
Bu deneyde görüntü işleme yordamlarını yapan RISC-V programlama projeleri geliştireceksin. Projeler, kimisi C’de kimisi çeviricide yazılmış, birden çok kaynak dosyaları içerecek. C işlevlerinin nasıl çevirici yordamlarını ya da çevirici işlevlerinin nasıl C yordamlarını çağırabileceğini göstereceğiz.

2. GÖRÜNTÜ İŞLEME ÖĞRETİCİSİ

Bu deneye RGB görüntüyü (Figür 1’in sol yanı) işleyip grilisini (Figür 1’in sağ yanı) oluşturan, bitmiş bir programı inceleyerek başlayacağız. Bu program C ile RISC-V çevirici dillerinde yazılıp PlatformIO ortamında çalışmak üzere yapılandırılmıştır. Şuradadır:

[RVfpgaPath]/RVfpga/Labs/Lab5/ImageProcessing

Kaynak kodu src alt dizinindedir.



Figür 1. RGB görüntünün Grili görüntüye dönüşümü.

A. Proje yapısı ile *main* işlevi

Program şu kaynak dosyalarından oluşur: **main.c**, **VanGogh_128.c**, **assemblySubroutines.S**. .c dosyaları işlevler (görüntü dönüşümlerini gerçekleştiren işlevler gibi) ile değişken bildirilerini (karakter dizisi olarak bildirilmiş görüntü gibi) barındırır. **assemblySubroutines.S** dosyası görseli rgb’den griliye dönüştüren şu çevirici dilinde gerçekleştirilmiş işlevi barındırır: *ColourToGrey_Pixel*.

Figür 2 projenin *main* işlevini gösterir. İlk girdi görüntü verisiyle N x M matris oluşturan *initColourImage* işlevini çalıştırır. Ardından renkli görüntüyü grili görüntüye dönüştürür (işlev *ColourToGrey*). Son olarak işlev bir ileti yazdırıp sonsuz döngüye girer (*while (1) ;*).

```

49  int main(void) {
50      // Create an NxM matrix using the input image
51      initColourImage(ColourImage);
52
53      // Transform Colour Image to Grey Image
54      ColourToGrey(ColourImage, GreyImage);
55
56      // Initialize Uart
57      uartInit();
58      // Print message on the serial output
59      printfNexys("Created Grey Image");
60
61      while(1);
62
63      return 0;
64  }

```

Figür 2. ImageProcessing projesindeki *main* işlevi

B. RGBli, grili görüntüler

Bir görüntü, elemanlarının belirli bir ölçekte piksel değerlerini gösterdiği bir piksel matrisinden oluşur. RGB’de bütün pikseler üç değere karşılık gelir, bu da kırmızı (**R**), yeşil (**G**), mavi (**B**) bileşen parlaklıklarıdır. Dolayısıyla renkli bir görüntünün pikselleri üç-bileşenli vektörler olacaktır. Bu projede RGB piksel türü için şu tanımlı kullanıyoruz:

```

typedef struct {
    unsigned char R;
    unsigned char G;
    unsigned char B;
} RGB;

```

Kod *RGB* adında bir yapı tanımlar. C’de, *struct* veri türü, bir adla belirlenen bir değişkenler, yüksek olasılıkla değişik türlerde, koleksiyonudur. Bu yapı aynı türden (*unsigned char*) *R*, *G*, *B* diye üç alan barındırır. Yani renk kanalları 8 bitle gösterilir, dolayısıyla piksel başına 24 bit olacak (24bpp) biçimde 256 ayrı yoğunluk düzeyi algılayabiliriz. Bu günümüzün dijital görüntü işleminde yaygın bir biçimdir.

Grili bir görüntüyü göstermek için 0 ile 255 arasında bir değer (bir kanal) piksellerin parlaklığını belirtir. Bu ImageProcessing projesinde grili görüntüyü bir 2-boyutlu karakter dizisiyle gösteriyoruz:

```

unsigned char GreyImage[N][M];

```

C. Renkli bir görüntüyü grili bir görüntüye dönüştürme

İki renk uzayı (RGB ile Grili) arasındaki dönüşüm şu ağırlık toplamayla yapılır:

$$\text{grey} = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

Bu denklemin tabanı şurada tanımlanan algoritmalarıdır

<https://www.mathworks.com/help/matlab/ref/rgb2gray.html>.

Pikselleri renk kanallarını denklemdaki ağırlıkla çarparak grili değerlerini hesaplıyoruz. Ağırlıkların toplamı (0.299+0.587+0.114) birdir, dolayısıyla sonuçtaki grili değer 0-255 aralığında olacaktır, yani bir bayt ile gösterilebilir.

Denklemden verilen ağırlıkları kullanmak için gerçek sayılarla işlememiz gerekir ancak **SweRV EH1** işlemcisi kayan noktalı sayı desteği içermez. İlk Kullanım Kılavuzunun Bölüm 5.H'indeki DotProduct programında gösterildiği gibi kayan nokta emülasyonu da kullanılabilir ancak bu deneyde tam sayı aritmetiği tabanlı bir yaklaşım kullanıyoruz. Ağırlıklar önce tam sayılara çevrilir, toplam ise bir ikinin üssü olur (burada 2^{10}). Ağırlıklar tam sayıya dönüştürmek için bütün kayan nokta ağırlıklarını 2^{10} ile çarpıp en yakın tam sayıya yuvarlıyoruz:

- $0.299 \cdot 2^{10} = 306.176 \approx 306$ (weight for R)
- $0.587 \cdot 2^{10} = 601.088 \approx 601$ (weight for G)
- $0.114 \cdot 2^{10} = 116.736 \approx 117$ (weight for B)

Son olarak grili değeri 0-255 aralığına indirgemek için toplamı 2^{10} ile bölüyoruz, ki bu değeri 10 bit sağa kaydırarak kolayca yapılabilir. Dolayısıyla son dönüşüm şu formülle elde edilir:

$$\text{grey} = (306 \cdot R + 601 \cdot G + 117 \cdot B) \gg 10$$

Önemli olarak sabitlerin toplamı (306+601+117) 1024 olduğu göz önünde bulundurulursa son grili değer yine 0-255 aralığında olacaktır.

Figür 3 *ColourToGrey* işleviyle (sol yan) *ColourToGrey* çalıştırdığı *ColourToGrey_Pixel* alt yordamının (sağ yan) görselleştirmesini yapar.



```

38  extern int ColourToGrey_Pixel(int R, int G, int B);
39
40  void ColourToGrey(RGB Colour[N][M], unsigned char Grey[N][M]) {
41      int i,j;
42
43      for (i=0;i<N;i++)
44          for (j=0;j<M;j++)
45              Grey[i][j] = ColourToGrey_Pixel(Colour[i][j].R, Colour[i][j].G, Colour[i][j].B);
46  }

```

```

1  .globl ColourToGrey_Pixel
2
3  .text
4
5  ColourToGrey_Pixel:
6
7      li x28, 306
8      mul a0, a0, x28
9
10     li x28, 601
11     mul a1, a1, x28
12
13     li x28, 117
14     mul a2, a2, x28
15
16     add a0, a0, a1
17     add a0, a0, a2
18
19     srl a0, a0, 10
20
21     ret
22
23 .end

```

Figür 3. *ColourToGrey* işlevi (*main.c* dosyasında gerçekleştirilmiştir) ile *ColourToGrey_Pixel* alt yordamı (*assemblySubroutines.S* dosyasında gerçekleştirilmiştir).

Çevirici dilinde semboller (değişkenler, işlevler, alt yordamlar) normalde yereldir, bir diğer deyişle diğer dosyalara görünmezdir. Bu yerel sembolleri genel sembollere dönüştürmek için `.globl` çevirici program yönlendirmesiyle dışa aktarmamız gerekir. Figür 3 satır (`.globl ColourToGrey_Pixel`) *ColourToGrey_Pixel* işlevini dışa aktararak farklı bir dosyada bulunan *ColourToGrey* işlevi tarafından kullanılabilir (*main.c*). Figür 3'ün sol yanındaki ilk satır (`extern int ColourToGrey_Pixel(int R, int G, int B)`) *ColourToGrey_Pixel* işlevini bu dosyanın dış işlevi olarak bildirir.


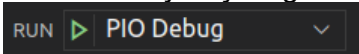
D. Programın yürütmesi, sonuçların görselleştirilmesi

Griliye dönüştürme tamamlandıktan sonra, programın yürütmesi sona ermeden önce, kimi bellek bölgelerinin içeriğini dosyalara dökebiliriz. Bunun için, GDB ayıklayıcısının dump komutunu kullanırız. Proje kodunu çalıştırmak ve görüntü sonuçlarını elde etmek için sonraki adımları izle:

1. VSCode ile PlatformIO'yu aç.
2. Üst menü çubuğunda, *File* (Dosya) → *Open Folder* (Klasörü Aç) tıklayıp ardından *[RVfpgaPath]/RVfpga/Labs/Lab5* dizinine git. *ImageProcessing* dizinini seç (açma, yalnızca seç) ardından pencerenin üstünde OK'a tıkla. PlatformIO şimdi projeyi açacaktır.
3. *platformio.ini* açıp *board_build.bitstream_file*'in yorumluğunu kaldırıp veri dosyasının dizinini gir. Örneğin Deney 1'de oluşturduğun veri dosyasını kullanabilirsin.

```
board_build.bitstream_file = [RVfpgaPath]/RVfpga/Labs/Lab1/Project1/Project1.runs/impl_1/rvfpga.bit
```

4. *src* dizinindeki bütün kaynak dosyalarını açıp (*main.c*, *assemblySubroutines.S*) çözümle ki programın nasıl çalıştığını anla.
5. RVfpga'i Nexys A7 kartına sol menü şeridindenki PlatformIO ikonuna tıklayarak indir, ardından *Project Tasks* → *env:swervolf_nexys* → *Platform genişletip Upload Bitstream* tıkla.
6. PlatformIO'da programı yürüt. Bunu kartta (bu durumda önce Nexys A7'ye, önceki adımda olduğu gibi RVfpga yüklemeniz gerekir) veya Whisper simülatörü (RVfpga İlk Kullanım Kılavuzunda açıklandığı gibi) kullanarak yapabilirsiniz. İki türlü

de PlatformIO'nun sol yan çubuğundaki "Run"  butonuna tıkla, ardından oynat butonuna  basarak ayıklayıcıyı başlat.

Yürütme *main* işlevinin yürütmesinin başında duracaktır, "Continue" butonuna

 basarak sürdür.

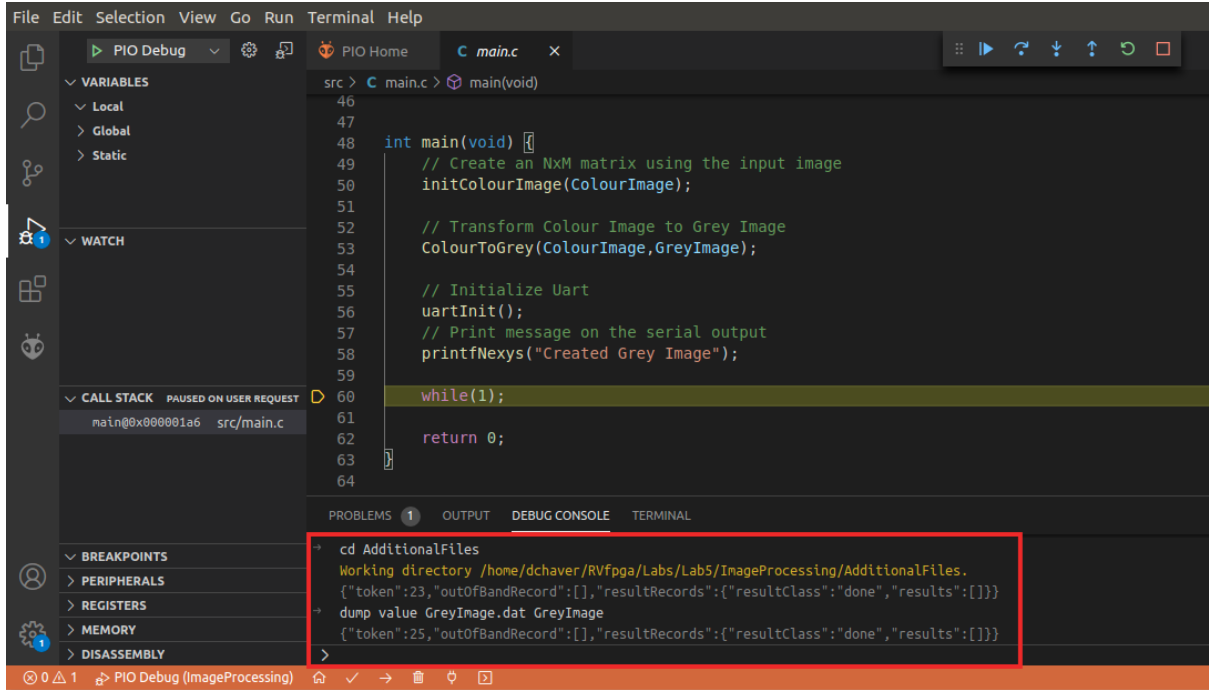
Kısa bir süre sonra (yaklaşık 1 saniye), program yukarıda açıklanan grili görüntü dönüşümlerini bitirip sondaki sonsuz (*while(1);*) döngüye erişecektir (Figür 2'ye

göz at). *Pause* butonuna basarak  yürütmeyi beklet.

7. Şu komutları *Debug Console*'da çalıştırarak grili görüntüyü (*GreyImage*) dışa aktar (Figür 4'e göz at, bu iki komutun yürütmesini gösterir):

```
cd AdditionalFiles
```

```
dump value GreyImage.dat GreyImage
```



Figür 4. Grili görüntüyü dosya olarak dışa aktar

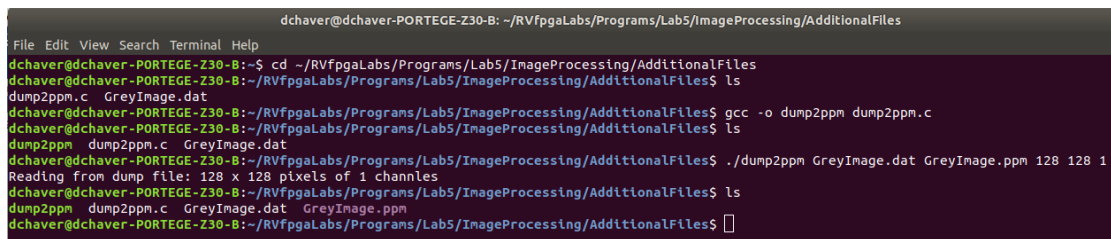
8. .dat dosyasını sisteminde görüntüleyebileceğin .ppm dosyasına dönüştür.

LINUX içerisinde: terminali açıp şu komutları yazarak yap (Figür 5'e göz at):

```
cd [RVfpgaPath]/RVfpga/Labs/Lab5/ImageProcessing/AdditionalFiles
```

```
gcc -o dump2ppm dump2ppm.c
```

```
./dump2ppm GreyImage.dat GreyImage.ppm 128 128 1
```



Figür 5. Görseli .ppm biçimine dönüştür

WINDOWS'ta: ya şöyle yap:

1. dump2ppm.exe yürütülebilirini şuradan kullanarak
`[RVfpgaPath] \RVfpga \Labs \Lab5 \ImageProcessing \AdditionalFiles`. Bir
komut istemcisi aç, o klasöre git, yürütülebilirini yukarıdaki argümanlarla çalıştır:

```
dump2ppm.exe GreyImage.dat GreyImage.ppm 128 128 1
```

Ya da şöyle

2. dump2ppm.c programını derlemek için Cygwin'i (RVfpga İlk Kullanım Kılavuzunda tanımlandığı gibi kurduysan) kullanarak. Ardından programı (dump2ppm.exe) Cygwin terminalinde ya da yukarıdaki birinci seçenekteki gibi komut istemcisinde çalıştır.

9. .ppm dosyasını GIMP, GNU Image Manipulation Program, kullanarak aç. Eğer program kurulu değilse kurucuyu indirmek için şu websitesine git:

<https://www.gimp.org/downloads/>

Grili görüntü Figür 1'in sağ yanındaki gibi görünmeli (renkli girdi görüntüsüne şuradan da erişebilirsin

`[RVfpgaPath]/RVfpga/Labs/Lab5/ImageProcessing/AdditionalFiles/VanGogh_128.ppm`, Figür 1'in sol yanındaki gibi görünmeli).

3. Alıřtırmalar

Alıřtırma 1. Programı bařka bir girdi görüntü üzerinde yürüt. řurada saęlanan görüntüyü kullanabilirsin: `[RVfpgaPath]/RVfpga/Labs/Lab5/ImageProcessing/src/TheScream_256.c` (Karřılık gelen `.ppm` görüntüsüne řuradan bakabilirsin: `[RVfpgaPath]/RVfpga/Labs/Lab5/ImageProcessing/AdditionalFiles/TheScream_256.ppm`. Bu görseli sen de `dat2ppm` programını çalıřtırarak oluřturacaksın, önceden tanımlandıęı gibi.)

Alıřtırma 2. *VanGogh* grili görüntüsündeki beyaza yakın (>235) ile siyaha yakın (<2) ögelerin sayısını sayan bir C iřlevi oluřtur. İki sayıyı da Western Digital'ın PSP ile BSP kütüphanelerini Deney 2'nin Bölüm 3'ünde açıklandığı gibi dizeysel konsolda yazdır.

Alıřtırma 3. **ColourToGrey_Pixel** çevirici altyordamını bir C iřlevine, **ColourToGrey** C iřlevini, **ColourToGrey_Pixel** C iřlevini çalıřtıran bir çevirici altyordamına dönüřtür.

- C'de bütün iřlevlerle genel deęiřkenler varsayılan olarak dıřarıya genel sembol olarak aktarılır, dolayısıyla *ColourToGrey_Pixel* iřlevini *ColourToGrey* altyordamında kullanabilirsin.
- Çevirici dilinde bir matrise eriřmek için bir ögenin adresini (i,j) dizinin bařlangıç adresine göre hesaplamalısın. ANSI C standardına göre iki-boyutlu diziler bellekte satır satır depolanır. Dolayısıyla satır i , sütun j 'deki bir pikselin adresi bařlangıç adresiyle bořluk $(i*M + j)*B$ toplanarak bulunur, burada M sütun sayısı, B ise piksel başına kaplanan bayt sayısıdır: RGB'de üç bayt, grilide bir bayt.

Alıřtırma 4. VanGogh renkli görüntüsüne bir **Bulanıklařtırma Filtersi** uygula (internette çok bilgi bulabilirsin; örneęin řuradaki bilgiyi kullanabilirsin: https://lodev.org/cgtutor/filtering.html#Find_Edges).

`.dat` görüntüsünü bir `.ppm` görüntüsüne dönüřtürmek için `dump2ppm` komut çalıřtırmada 1 kanal yerine 3 kanalın deęerlendirilmesi için bir yeri deęiřtirmen gerek:

```
./dump2ppm FilterColourImage.dat FilterColourImage.ppm 128 128 3
```

Dahası, filtreli görseli orijinaliyle karřılařtırabilirsin, řuradan eriřilebilir `[RVfpgaLabsPath]/RVfpgaLabs/Programs/Lab5/ImageProcessing/AdditionalFiles/VanGogh_128.ppm`