SENG440 Embedded Systems

– Lesson 103: Color Space Conversion –

Mihai SIMA

msima@ece.uvic.ca

Academic Course

Copyright © 2023 Mihai SIMA

Disclaimer

The purpose of this course is to present general techniques and concepts for the analysis, design, and utilization of embedded systems. The requirements of any real embedded system can be intimately connected with the environment in which the embedded system is deployed. The presented design examples should not be used as the full design for any real embedded system.

Lesson 103: Color Space Conversion

Trichromatic Theory

- **Trichromatic Theory** $\rightarrow$ all the colors in the visible spectrum can be matched by appropriately mixing three **primary colors**
- It is not important which primary colors are used as long as mixing two of them does not produce the third color
- Red-Green-Blue (RGB) system is used in displays that emit light
- The nonlinearity of the old (Cathode-Ray Tube) CRT monitors is compensated by applying a nonlinear transfer function to RGB intensities to form Gamma–Corrected Red, Green, and Blue (R'G'B')
- A color space is a mathematical representation of a set of colors
- Several standard color spaces: R'G'B', Y'CC, Y'UV
- Y'CC and Y'UV are used in video standards
- Good books by Charles Poynton (see References)

Why we need color space conversion

- The space RGB: each component represents a color
- The human eye is less sensitive to color than luminance
- To reduce the storage requirements and/or transmission rate
  - Transmit luminance with full resolution
  - Represent the color information with lower resolution
    - Reduce the resolution when converting from RGB representation to Luminance+Color representation
    - Increase the resolution when converting from a Luminance+Color representation to RGB representation
- Reduce the resolution: **downsampling**
  - Discard samples – which ones?
- Increase the resolution: **upsampling**
  - Create new samples – how?

Luminance and Chrominance

- Consider that R', G', and B' are in the range $[0 \cdots +1.0]$
- **Luma** signal (represents luminance or brightness):

$$Y' = 0.299R' + 0.587G' + 0.114B'$$

- Luma contains a large fraction of the green information
- Two color difference components with no contribution from luminance:

$$B' - Y' = -0.299R' - 0.587G' + 0.886B'$$

$$R' - Y' = 0.701R = -0.587G' - 0.114B'$$

- Matrix notation (be very curious and calculate the condition number!)

$$\begin{pmatrix} Y' \\ B' - Y' \\ R' - Y' \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.299 & -0.587 & 0.886 \\ 0.701 & -0.587 & -0.114 \end{pmatrix} \cdot \begin{pmatrix} R' \\ G' \\ B' \end{pmatrix}$$

Color Space Conversion – $R'G'B'$-to-$Y'P_BP_R$ I

- **Analog video equipment** – $Y'P_BP_R$ are defined as follows:
  - Y' ranges $[0 \cdots +1.0]$
  - P_B and P_R range $[-0.5 \cdots +0.5]$
- To construct $Y'P_BP_R$ from the basic Y', $(B' - Y')$, and $(R' - Y')$
  - Scale the $(B' - Y')$ row by $\frac{0.5}{1-0.114} = \frac{0.5}{0.886}$
  - Scale the $(R' - Y')$ row by $\frac{0.5}{1-0.299} = \frac{0.5}{0.701}$

$$\begin{pmatrix} Y' \\ P_B \\ P_R \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{pmatrix} \cdot \begin{pmatrix} R' \\ G' \\ B' \end{pmatrix}$$

- Inverse transform: **http://www.poynton.com/ColorFAQ.html**

Color Space Conversion – $R'G'B'$-to-$Y'P_BP_R$ II

- Summary: the $R'G'B'$-to-$Y'P_BP_R$ transformation assumes:
  - R', G', and B' range $[0 \cdots +1.0]$
  - Y' ranges $[0 \cdots +1.0]$
  - P_B and P_R range $[-0.5 \cdots +0.5]$
  - The matrix contains fractional numbers
- We need to use only integer arithmetic!
  - R', G', and B' can be, for example, 8-bit unsigned integers
  - Y' can also be an 8-bit unsigned integer
  - How many bits do we need to represent the matrix elements?
- Saturating arithmetic is needed!
  - R', G', B', and Y' range $[0 \ldots 255]$, P_B and P_R range $[-128 \ldots 127]$
  - Hardware-based solution: make sure the hardware will saturate the result
  - Software-based solution on a 32-bit processor: can we use saturating operations to implement 8-bit saturating arithmetic?
- It is always a good idea to check the condition number

Color Space Conversion – $R'G'B'$-to-$Y'C_BC_R$

- There are many standards for digital versions of this matrix
- Recommendation ITU-R BT.601-4 = the international standard for studio-quality component digital video
- Luminance Y':
  - Coded in 8 bits
  - Excursion of 219 and an offset of 16 (range of $[+16 \ldots 235]$)
  - The extremes of the coding range provide headroom and footroom for accomodation of ringing from filters
- Chrominance C_B and C_R
  - Coded in 8 bits
  - Excursion of ± 112 and offset of $+128$ (range of $[+16 \ldots 240]$)

Direct Color Space Conversion – $R'G'B'$-to-$Y'C_BC_R$

- To form $Y'C_BC_R$ from Y', $B'-Y'$, $R'-Y'$ in the range $[0\cdots+1.0]$

$$Y' = 16 + \text{round}\left(219\,Y'\right)$$

$$C_B = 128 + \text{round}\left\{112\left[\frac{1}{1-0.114}(B'-Y')\right]\right\}$$

$$C_R = 128 + \text{round}\left\{112\left[\frac{1}{1-0.299}(R'-Y')\right]\right\}$$

- Matrix form: scale the rows by the factors 219, 224, and 224

$$\begin{pmatrix} Y' \\ C_B \\ C_R \end{pmatrix} = \begin{pmatrix} 16 \\ 128 \\ 128 \end{pmatrix} + \begin{pmatrix} 65.481 & 128.553 & 24.966 \\ -37.797 & -74.203 & 112.0 \\ 112.0 & -93.786 & -18.214 \end{pmatrix} \cdot \begin{pmatrix} R' \\ G' \\ B' \end{pmatrix}$$

- C_B and C_R will be downsampled

Direct Color Space Conversion – $R'G'B'$-to-$Y'C_BC_R$

$$\begin{cases} Y' - 16 = +0.257R' + 0.504G' + 0.098B' \\ C_B - 128 = -0.148R' - 0.291G' + 0.439B' \\ C_R - 128 = +0.439R' - 0.368G' - 0.071B' \end{cases}$$

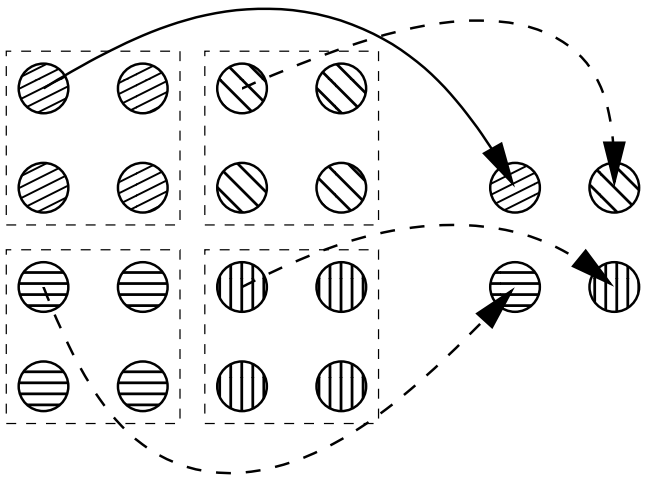
- Y', C_B, C_R, R', G', and B' are 8-bit integers
- For each C_R (C_B) chroma samples there are four luma values – we have to do **downsampling**
- Downsampling: four chrominance samples (C_R, C_B) are replaced by a single chrominance (C_R, C_B) sample
  - Approach 1: discard three samples and keep one
  - Approach 2: **calculate the average of the four samples**
  - Approach 3: more complex filtering

Color Space Conversion – Downsampling I

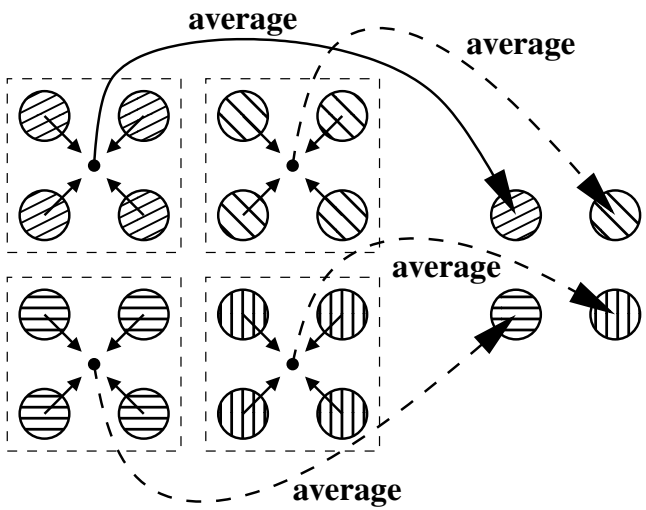
- Reduce the spatial resolution of chrominance signals to half
- Downsampling is carried out on both horizontal and vertical dimensions
  - Four C_R (or C_B) samples are replaced by a single C_R (or C_B) sample
  - The resulting image will be four times smaller than the initial one
- How to filter out every other pixel?
  - Programming without DSP: **discard pixels** – this translates into a poor image quality
  - Programming with minimal DSP: **the resulting pixel is the average of four pixels** – slightly better image quality
  - Programming with intensive DSP: **use Filter Theory**, since reducing the sampling rate is essentially a filtering problem – good image quality
- Be very curious and try the last approach!

Color Space Conversion – Downsampling II

- A graphical representation of the downsampling process is shown below



Two-dimensional downsampling by discarding pixels



Two-dimensional downsampling by averaging pixels

Inverse Color Space Conversion – $Y'C_BC_R$-to-$R'G'B'$ I

$$\begin{cases} R' = 1.164(Y' - 16) + 1.596(C_R - 128) \\ G' = 1.164(Y' - 16) - 0.813(C_R - 128) - 0.391(C_B - 128) \\ B' = 1.164(Y' - 16) + 2.018(C_B - 128) \end{cases}$$

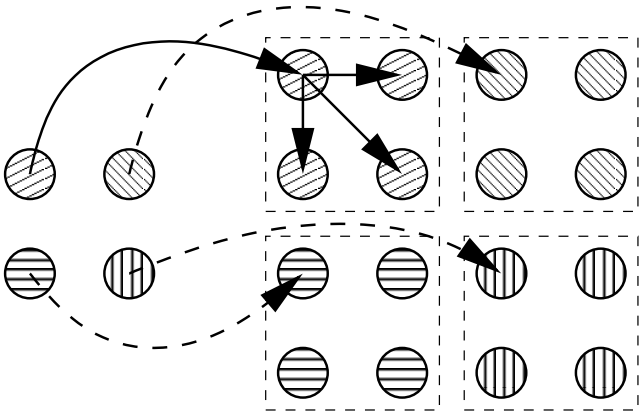
- Y', C_B, C_R, R', G', and B' are 8-bit integers
- For each C_R and C_B chroma samples there are four luma samples (because the human eye is less sensitive to color than luminance) – we have to do **upsampling**, e.g., by replication
- Upsampling: one chrominance sample (C_R, C_B) is replaced by four chrominance (C_R, C_B) samples
  - Approach 1: replicate one sample into four samples
  - Approach 2: **use interpolation to generate extra samples**
  - Approach 3: more complex filtering

Color Space Conversion – Upsampling I

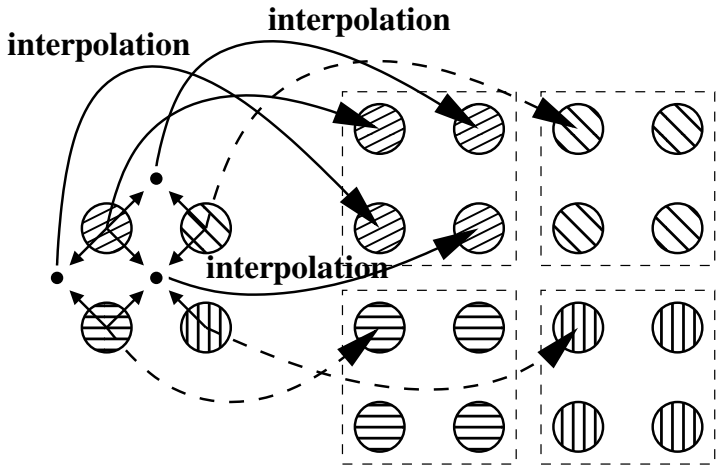
- Double the spatial resolution of chrominance signals
- Upsampling is carried out on both horizontal and vertical dimensions
  - One C_R (or C_B) value is replaced by four C_R (or C_B) values
  - The resulting image will be four times larger than the initial one
- How to create new pixels?
  - Programming without DSP: **replicate pixels** – this translates into a poor image quality
  - Programming with minimal DSP: **generate the new pixels by linear interpolation** – slightly better image quality
  - Programming with intensive DSP: **use the Filter Theory**, since increasing the sampling rate is a filtering problem – good image quality
- Be very curious and try the last approach!

Color Space Conversion – Upsampling II

- A graphical representation of the upsampling process is shown below



Two-dimensional upsampling by replicating pixels



Two-dimensional upsampling by interpolation

Software for Color Space Conversion I

In the directory **`LESSON_103_CSC`** the following routines are provided as a starting point for the design and optimization process:

- **`CSC_main.c`** – includes the **`main()`** function
- **`CSC_global.h`** – header file
- **`CSC_RGB_to_YCC.c`** – both floating point and fixed point functions
- **`CSC_YCC_to_RGB.c`** – both floating point and fixed point functions
- **`rgb_to_ycc_and_back.m`** – Octave / MATLAB® script to convert individual pixels
- **`image_input_64_48.xcf`** – sample 64-by-48 image file in **`xcf`** (GIMP) format
- **`image_input_64_48.raw`** – sample 64-by-48 image file in **`raw`** (i.e., 24-bit RGB without header) format

Software for Color Space Conversion II

First, the software breaks the input file **`image_input_64_48.raw`** into the color components (Red, Green, Blue) and saves them into three separate files:

- **`image_echo_R_64_48.raw`**
- **`image_echo_G_64_48.raw`**
- **`image_echo_B_64_48.raw`**

Next, the software calculates the luminance and chrominance components and saves them into three separate files:

- **`image_output_Y_64_48.raw`**
- **`image_output_Cb_64_48.raw`**
- **`image_output_Cr_64_48.raw`**

Software for Color Space Conversion III

Then, the software converts the YCC representation back to an RGB representation, and saves the color components into three files:

- **`image_output_R_64_48.raw`**
- **`image_output_G_64_48.raw`**
- **`image_output_B_64_48.raw`**

The software assembles the color components into an RGB raw file:

- **`image_output_RGB_64_48.raw`**

The **`raw`** files can be visualized with **`GIMP`** (**`DarkTable`** or **`RAWTherapee`** plugin need to be installed)

Software for Color Space Conversion IV

- It is recommended to first try the conversion in floating-point arithmetic. For this, the file **CSC_global.h** needs to be edited in order to set:

```
#define RGB_to_YCC_ROUTINE 1
#define YCC_to_RGB_ROUTINE 1
```

- To compile on a workstation:

```
gcc CSC_main.c CSC_RGB_to_YCC.c
CSC_YCC_to_RGB.c -o CSC.out
```

- To execute:

```
./CSC.out
```

- To check that the code is functionally correct visualize the **image_input_64_48.raw** and **image_output_64_48.raw** files with **GIMP**

Floating-Point Implementation of Color Space Conversion I

RGB to YCC (The direct CSC transformation)

- The pseudocode is shown below

```
float r, g, b, y, cr, cb;
int main( void) {
  for( ... all rows in an image ...)
    for( ... all columns in an image ...) {
      ... read r, g, b ...
      y = 16.0 + 0.257 r + 0.504 g + 0.098 b
      cb = 128.0 - 0.148 r - 0.291 g + 0.439 b
      cr = 128.0 + 0.439 r - 0.368 g - 0.071 b
      ... do this conversion 4 times ...
      ... average four cb/cr values to do downsampling ...
    }
  return( 1);
}
```

Floating-Point Implementation of Color Space Conversion II

RGB to YCC (The direct CSC transformation)

- Compile this code on ARM: each float operation is compiled into a large number of instructions (entire routines!)
  - Reason: there is no floating-point unit
- On an embedded platform this is clearly way too slow!
  - Strategy: trade off dynamic range for computing time
- The *float* arithmetic needs to be converted to *integer* arithmetic
  - 0.257 will become an integer
  - 16.0 will also become an integer – think twice if you want to represent the real value 16.0 as integer 16
- **Difficulty**: the large dynamic range to be covered
  - The smallest real value: 0.071
  - The largest value: 128.0

Fixed-Point Implementation of Color Space Conversion I

Work Plan

- All operations will be implemented in fixed-point arithmetic. This requires that all arguments are integers
- Due to the reduced wordlength (say, $K = 8$ or $K = 12$ bits), it is important to implement rounding
- To guarantee that overflow and underflow are never encountered, saturating arithmetic should be considered
- Since the application exhibits data-level parallelism, vector (SIMD) operations are good candidates for optimization
- Since color space conversion is a data-dominant application, the pattern of accessing cache is a good candidate for optimization
- To achieve an accurate image conversion different techniques for chrominance downsampling / upsampling will be considered

Fixed-Point Implementation of Color Space Conversion I

RGB to YCC (The direct CSC transformation)

- Since all input and output samples (R, G, B, Y, C_B, and C_R) are 8-bit integers, it has been decided that the bitwidth is $K = 8$ bits
- Temporary variables may have a larger bitwidth (e.g., 16-bit, or 32-bit)
- Regarding the direct CSC transformation it is observed that:
  - $\min\left(128 - 0.148\,R - 0.291\,G\right) = \min\left(128 - 0.148 \cdot 255 - 0.291 \cdot 255\right) = 16 > 0$
  - $\min\left(128 - 0.368\,G - 0.071\,B\right) = \min\left(128 - 0.148 \cdot 255 - 0.291 \cdot 255\right) = 16 > 0$
- Thus, **signed representation is not needed**; unsigned representation suffices (no bit devoted to sign; all bits devoted to magnitude)

Fixed-Point Implementation of Color Space Conversion II

RGB to YCC (The direct CSC transformation)

- **Is saturating arithmetic needed?**
- The largest possible values are:

$$\max Y = 16 + 0.257 \cdot 255 + 0.504 \cdot 255 + 0.098 \cdot 255 = 235$$
$$\max C_B = 128 - 0.148 \cdot 0 - 0.291 \cdot 0 + 0.439 \cdot 255 = 240$$
$$\max C_R = 128 + 0.439 \cdot 255 - 0.368 \cdot 0 - 0.071 \cdot 0 = 240$$

- The smallest possible values are:

$$\min Y = 16 + 0.257 \cdot 0 + 0.504 \cdot 0 + 0.098 \cdot 0 = 16$$
$$\min C_B = 128 - 0.148 \cdot 255 - 0.291 \cdot 255 + 0.439 \cdot 0 = 16$$
$$\min C_R = 128 + 0.439 \cdot 0 - 0.368 \cdot 255 - 0.071 \cdot 255 = 16$$

- It is obvious that overflow and underflow will never occur. As a result, **saturating arithmetic is not needed**

Fixed-Point Implementation of Color Space Conversion III

RGB to YCC (The direct CSC transformation)

- The largest coefficient is $0.504 < 1.0$, which means that the scale factor is given by $\frac{2^K}{1.0} = 2^8$
- Conversion of matrix elements to Integer representation

$$
\begin{aligned}
c_{11} = 0.257 &\rightarrow C_{11} = \text{round}\left(2^8 \cdot 0.257\right) = 66 \\
c_{12} = 0.504 &\rightarrow C_{12} = \text{round}\left(2^8 \cdot 0.504\right) = 129 \\
c_{13} = 0.098 &\rightarrow C_{13} = \text{round}\left(2^8 \cdot 0.098\right) = 25 \\
c_{21} = 0.148 &\rightarrow C_{21} = \text{round}\left(2^8 \cdot 0.148\right) = 38 \\
c_{22} = 0.291 &\rightarrow C_{22} = \text{round}\left(2^8 \cdot 0.291\right) = 74 \\
c_{23} = 0.439 &\rightarrow C_{23} = \text{round}\left(2^8 \cdot 0.439\right) = 112 \\
c_{31} = 0.439 &\rightarrow C_{31} = \text{round}\left(2^8 \cdot 0.439\right) = 112 \\
c_{32} = 0.368 &\rightarrow C_{32} = \text{round}\left(2^8 \cdot 0.368\right) = 94 \\
c_{33} = 0.071 &\rightarrow C_{23} = \text{round}\left(2^8 \cdot 0.071\right) = 18
\end{aligned}
$$

Fixed-Point Implementation of Color Space Conversion IV

RGB to YCC (The direct CSC transformation)

- For example, the calculation of the luminance in fixed-point arithmetic is:

```
#define K 8
Y_temp = 16
       + (((C11 * (int)R) + (1 << (K-1))) >> K)
       + (((C12 * (int)G) + (1 << (K-1))) >> K)
       + (((C13 * (int)B) + (1 << (K-1))) >> K);
Y = (uint8_t)Y_temp;
```

- Since the multiplication of two 8-bit unsigned bytes generates a 16-bit unsigned integer, the Red, Green, and Blue bytes are first promoted to 16-bit (or wider) unsigned integers through cast operators
- Rounding to the nearest integer is used: `(...+ (1 << (K-1))) >> K`
- There is no danger of overflow / underflow, as it was discussed
- Finally, a cast to an 8-bit unsigned byte is performed

Fixed-Point Implementation of Color Space Conversion V

RGB to YCC (The direct CSC transformation)

- It is reminded that the multiplication of two 8-bit unsigned integers generates a 16-bit unsigned integer
- The 16-bit products are converted into 8-bit integers through rounding before the four-operand addition is performed
- The precision can be slightly improved by performing the addition with 16-bit arguments followed by rounding the sum into an 8-bit integer

```
#define K 8
Y_temp = (16 << K) + (C11 * (int)R)
       + (C12 * (int)G) + (C13 * (int)B);
Y = (uint8_t)((Y_temp + (1 << (K-1))) >> K);
```

- This avenue is not addresed any further, but students are encouraged to investigate it (especially if vector operations are to be used)

Fixed-Point Implementation of Color Space Conversion VI

RGB to YCC (The direct CSC transformation)

- The (non-optimized) code with 8-bit addition is shown in file **`CSC_RGB_to_YCC.c`**
- To activate the fixed-point arithmetic the file **`CSC_global.h`** needs to be edited in order to set:
  `#define RGB_to_YCC_ROUTINE 2`
- To compile on a workstation:
  `gcc CSC_main.c CSC_RGB_to_YCC.c CSC_YCC_to_RGB.c -o CSC.out`
- To execute:
  `./CSC.out`
- To check the result visualize the **`image_input_64_48.raw`** and **`image_output_64_48.raw`** files with **`GIMP`**

Fixed-Point Implementation of Color Space Conversion I

YCC to RGB (The inverse CSC transformation)

- Since all input and output samples (R, G, B, Y, C_B, and C_R) are 8-bit integers, it has been decided that the bitwidth is $K = 8$ bits
- Temporary variables may have a larger bitwidth (e.g., 16-bit or 32-bit)
- It is reminded that $16 \leq Y \leq 235$ and $16 \leq C_B, C_R \leq 240$
- It is observed that:
  - In the RGB domain all $256 \cdot 256 \cdot 256 = 16,777,216$ data triplets are valid, since each color ranges from 0 to 255
  - In the YCC domain not all data triplets are valid For example, it is not possible to have no luminance ($Y = 16$) with maximum chrominance ($C_B = 240$ and $C_R = 240$) in a signal, since this would correspond to $R = 179$, $G = -135$ (negative!), and $B = 226$

Fixed-Point Implementation of Color Space Conversion II

YCC to RGB (The inverse CSC transformation)

- The **`rgb_to_ycc.m`** script can be used to convert pixels from RGB domain to YCC domain and back. Conversion examples:

No.	RGB	$\longrightarrow$	YCC	$\longrightarrow$	RGB
1	$(0, 0, 0)$	$\longrightarrow$	$(16, 128, 128)$	$\longrightarrow$	$(0, 0, 0)$
2	$(255, 255, 255)$	$\longrightarrow$	$(235, 128, 128)$	$\longrightarrow$	$(255, 255, 255)$
3	$(255, 0, 0)$	$\longrightarrow$	$(82, 90, 240)$	$\longrightarrow$	$(256, 1, 0)$
4	$(0, 255, 0)$	$\longrightarrow$	$(145, 54, 34)$	$\longrightarrow$	$(0, 256, 1)$
5	$(0, 0, 255)$	$\longrightarrow$	$(41, 240, 110)$	$\longrightarrow$	$(0, 0, 255)$
6	$(255, 255, 0)$	$\longrightarrow$	$(210, 16, 146)$	$\longrightarrow$	$(255, 255, 0)$
7	$(255, 0, 255)$	$\longrightarrow$	$(107, 202, 222)$	$\longrightarrow$	$(256, 1, 255)$
8	$(0, 255, 255)$	$\longrightarrow$	$(107, 202, 222)$	$\longrightarrow$	$(1, 255, 256)$
9	$(121, 107, 200)$	$\longrightarrow$	$(121, 167, 128)$	$\longrightarrow$	$(122, 107, 201)$

Fixed-Point Implementation of Color Space Conversion III

YCC to RGB (The inverse CSC transformation)

- The blue figures indicate arithmetic error.
- The red figures indicate arithmetic overflow / underflow.
- Since there are red figures $\longrightarrow$ **saturating arithmetic is needed**
- Unsigned representation is used
- The largest coefficient is $2.018 < 4.0$, which means that the scale factor is given by $\frac{2^K}{4.0} = 2^6$
- Conversion of matrix elements to Integer representation

$$
\begin{aligned}
d_1 = 1.164 \quad &\rightarrow \quad D_1 = \text{round}\left(2^6 \cdot 1.164\right) = 74 \\
d_2 = 1.596 \quad &\rightarrow \quad D_2 = \text{round}\left(2^6 \cdot 1.596\right) = 102 \\
d_3 = 0.813 \quad &\rightarrow \quad D_3 = \text{round}\left(2^6 \cdot 0.813\right) = 52 \\
d_4 = 0.391 \quad &\rightarrow \quad D_4 = \text{round}\left(2^6 \cdot 0.391\right) = 25 \\
d_5 = 2.018 \quad &\rightarrow \quad D_5 = \text{round}\left(2^6 \cdot 2.018\right) = 129
\end{aligned}
$$

Fixed-Point Implementation of Color Space Conversion IV

YCC to RGB (The inverse CSC transformation)

- For example, the calculation of Green in fixed-point arithmetic is:

```
#define K 8
G_temp = (((D1 * (int)(Y-16))    + (1 << (K-1))) >> K)
       + (((D3 * (int)(Cr-128)) + (1 << (K-1))) >> K)
       + (((D4 * (int)(Cb-128)) + (1 << (K-1))) >> K);
G = (uint8_t)G_temp;
```

- Since the multiplication of two 8-bit unsigned bytes generates a 16-bit unsigned integer, the Y, C_B, and C_R, bytes are first promoted to 16-bit unsigned integers through cast operators
- Rounding to the nearest integer is used after each multiplication to reduce the representation to 8-bit integers
- Overflow can be encountered; thus, **saturating arithmetic is needed**
- Finally, a cast to an 8-bit unsigned byte is performed

Fixed-Point Implementation of Color Space Conversion V

YCC to RGB (The inverse CSC transformation)

- Similar with the direct transformation, the precision can be slightly improved by performing the addition with 16-bit arguments followed by rounding the sum into an 8-bit integer:

```
#define K 8
G_temp = (D1 * (int)(Y-16))
       + (D3 * (int)(Cr-128))
       + (D4 * (int)(Cb-128));
G = (uint8_t)((G_temp + (1 << (K-1))) >> K) ;
```

- This avenue is not addresed any further, but students are encouraged to investigate it (especially if vector operations are to be used)

Fixed-Point Implementation of Color Space Conversion VI

YCC to RGB (The inverse CSC transformation)

- The (non-optimized) code is shown in the file **`CSC_YCC_to_RGB.c`**
- To activate the fixed-point arithmetic the file **`CSC_global.h`** needs to be edited in order to set:
  `#define YCC_to_RGB_ROUTINE 2`
- To compile on a workstation:
  `gcc CSC_main.c CSC_RGB_to_YCC.c CSC_YCC_to_RGB.c -o CSC.out`
- To execute:
  `./CSC.out`
- To check result visualize the **`image_input_64_48.raw`** and **`image_output_64_48.raw`** files with **`GIMP`**

Generating ARM Code I

- The cross compiler **`arm-none-eabi-gcc`** version 7.5.0 running under OpenSUSE Linux has been used to compile the C code.
  - A '**`none`**' compiler is used for building applications to run without OS
  - Standard library functions (such as **`scanf`** and **`printf`**) as well as header files (such as **`stdio.h`**) may not be available
- The target processor is ARM920T (its architecture is ARMv4T)
  - ARM920T has been widely used in industry for many years
  - For example, Samsung S3C2440A single-chip embedded microcontroller includes an ARM920T core
- The command line is given below
  - The '**`-S`**' flag instructs the compiler to save the assembly file

```
linux> arm-none-eabi-gcc -mcpu=arm920t -S \
-ffreestanding CSC_RGB_to_YCC_01.c
```

Generating ARM Code II

- Why the compilation option **`-ffreestanding`** is needed? Students are required to read the **`gcc`** documentation and answer this question
- Students will observe that the floating-point routine **`CSC_RGB_to_YCC_brute_force_float()`** generates lots of calls to floating-point functions, which increases the execution time
- Students will observe that the fixed-point routine **`CSC_RGB_to_YCC_brute_force_int()`** generates integer arithmetic instructions with many **`LOAD`** and **`STORE`** intructions
- The same analysis can be carried out for the dual routines **`CSC_YCC_to_RGB_brute_force_float()`** and **`CSC_YCC_to_RGB_brute_force_int()`**

Fixed-Point Implementation: Opportunities for Optimization

- Reduce the number of cache misses by properly accesing data stored in memory
- Reduce the traffic with the memory by the use of:
  - Local variables rather than global variables
  - Vector registers **D** (if available) as a scratchpad memory, since accessing any of them does not generate cache misses
- Use vector operations to process pixels in parallel
  - This approach requires the use of intrinsic / built-in operations
- Try the use of truncation rather than rounding
  - This approach slightly improves the speed at the expense of precision (provided that the processor does not have dedicated hardware to implement rounding)

Color Space Conversion – hardware-based solution I

- Assume that the optimized all-software solution is still too slow
- Investigate hardware support for computationally-demanding operations
- **What to support in hardware?**
  - The entire CSC matrix transformation (all three lines)?
  - Only one line (color) at a time? $\longrightarrow$ three new instructions are needed
- Limitations due to the architecture of the processor (ARM in this case)
  - Maximum two input arguments and one result per instruction are allowed
  - More than two arguments needed? Additional dummy instructions needed to upload extra arguments to the functional unit
  - More than one result needed? Additional dummy instructions needed to donwload the extra result(s) from the functional unit
  - **Packing** the arguments and/or results when their wordlength is not large (as it is the case in color space conversion)

Color Space Conversion – hardware-based solution II

- Assume the packing strategy
- Downsampling can be carried out in hardware, too! Think about that.

```
int r, g, b, y, cr, cb;
int main( void)
  for( ... all rows in an image ...)
    for( ... all columns in an image ...)
      ... read r, g, b ...
      ... pack r, g, b ...          // << This is overhead!
      CALL_HARDWARE ( r, g, b, y, cb, cr)
      ... unpack y, cb, cr ...      // << This is overhead!
      ... do this conversion 4 times ...
      ... average four cb/cr values to do downsampling ...

  exit( 1);
```

Color Space Conversion – project requirements

- Build a testbench containing an image (the image should be large enough in order not to fit into the cache)
- Design a color space conversion algorithm using only integer arithmetic
  - Recall that you should use saturating arithmetic
- Provide a all-software solution and estimate its performance
  - Keep an eye on cache misses
- Provide architectural support for multiplication-by-constant operations
  - Define new instructions
  - Implement the new instructions in hardware and/or firmware
  - Rewrite the code in order to instantiate the new instructions
- Compare the hardware and/or firmware solutions with the all-software solution

References

- Charles Poynton, *Digital Video and HD – Algorithms and Interfaces*, Morgan Kaufmann, 2012.
- ***, *Understanding Color Spaces and Color Space Conversion*, The MathWorks, Inc., 2023, `https://www.mathworks.com/help/images/understanding-color-spaces-and-color-space-conversion.html`
- Marko Tkalčič and Jurij F. Tasič, *Color Spaces: Perceptual, Historical, and Application Background*, August 2003.

Project Specification Sheet

Summer 2023

- **Student name:** .....
- **Student ID:** .....
- **Function to be optimized:** .....Color Space Conversion
- **Language:** .....C/C++
- **Processor:** .....32-bit ARM
- **Wordlength:** .....8 bits / pixel
- **Deadline:** .....

Questions, feedback

Notes I

Notes II

Notes III