```
// Name        : Embbeded_FACE_DETECTION.c

/*
    This file is part of IP_Embedded_Int_ReduceIntegral_LBP.
    Copyright (C) 2014, Laurentiu ACASANDREI

    IP_Embedded_Int_ReduceIntegral_LBP is free software: you can
    redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    IP_Embedded_Int_ReduceIntegral_LBP is distributed in the
    hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with IP_Embedded_Int_ReduceIntegral_LBP.  If not,
    see <http://www.gnu.org/licenses/>.


    Contact      : laurentiu@imse-cnm.csic.es
                   www.imse-cnm.csic.es

    Disclaimer   : All information is provided "as is", there is no warranty that
                   the information is correct or suitable for any purpose,
                   neither implicit nor explicit.
    Description : This is the optimized algorithmic version using the MB-LBP
                  hardware accelerator. For this purpose the integral elements
                  uint16_t are reversed(because the data is accessed on word).
                  IT contains debug code that verifies is the integral is correctly
                  reversed(see imse_setImageEmbedd_v4()).
*/

#include "Target_Config.hpp"
#include "Embedd_LBPClassifierData_Integer.hpp"
#include "IP_LBP_esential.hpp"

#include "Embedd_aux_LBP_func_luismi.hpp"
#include "Imse_vga_driver.h"

#include <iostream>
#include <stdio.h>
#include <time.h>
#include <stdint.h> // type uint8_t, ...


#include "800x600_color.h"

//static int imse_MatType=1111638032;

#if LEON3_UART_CONTROLLED
    #include "laur_apbuart.hpp"
    #include "Leon3mp_Serial_Driver.hpp"
#endif

#if LEON3_UART_CONTROLLED
    static void ScaleImage_SW( imse_Mat raw_frame, int height, int width );
    static void ScaleImage_HW( imse_Mat* src_image, imse_Mat* dst_image, int width, int
```

```
height);
    int prepare_picture_for_svga(unsigned int** video_address, imse_Mat img);
    static unsigned char __attribute__ ((aligned (512))) memory_data [2048 * 2048]={99};
    static unsigned char __attribute__ ((aligned (512))) data_out [2048 * 2048]={100};
#else
int prepare_color_image_for_svga (unsigned int** video_address, imse_Mat* src_img);
int prepare_gray_image_for_svga (unsigned int** output_image, imse_Mat* src_img);
int fill_gray_for_svga (unsigned int* buffer, imse_Mat* src_img, int buffer_rows, int
buffer_cols);
int fill_color_for_svga (unsigned int* buffer, imse_Mat* src_img, int buffer_rows, int
buffer_cols);
static void New_ScaleImage_SW (imse_Mat* src_image, imse_Mat* dst_image);
static void New_ScaleImage_HW (imse_Mat* src_image, imse_Mat* dst_image);
int display_width = 800, display_height = 600;
double total_time = 0, secs = 0;
#endif

#if DISPLAY_PICTURE
    int svgactrl_test(unsigned int addr, unsigned int real, unsigned int alloc,
                      unsigned int fbaddress, int format, int delay, int blank);
    int disable_vga(unsigned int addr);
#endif

int main(int argc, const char** argv )
{
    printf("Start Leon3 \n");

    unsigned int *img_dynamic = 0, *prepared_vga_img = 0;
    img_dynamic = (unsigned int*) malloc (display_width * display_height * sizeof (unsigned
int));

    #if LEON3_UART_CONTROLLED

        intialize_cmd_parameters_index();
        is_end = 0;
        uint8_t end_of_detection=0xFF;
        ReceiverState = Idle;
        unsigned int value;

        // IP init  ****************************
        //imse_init_leon3_shared_mem(); // init Ip LBP shared mem with the classifier
        catch_interrupt(irqhandler_IP, 10);  // We initialize the interrupt
        /// end IP in it

        UART_DISABLE(BASE_ADRR_UART1);

        //Clearing the receiver fifo before receiving any important info
        // If the receiver fifo is not clear the program is blocked.
        //clearing_uart_fifo_and_status(); ///!!!!!! Attention. Enabling this can break your
execution
        // Initializing the uart

        value=UART_ENABLE(BASE_ADRR_UART1, ENABLE_TX | ENABLE_RX | RX_unsigned_int
,BAUD_RATE);
        printf("\n IP Embedd Int Reduced minSize 24x24:  Set uart1 in Tx and \n RX mode with
BAUD_RATE=%d \n",BAUD_RATE); // This is displayed in the jtga debug mode
        if (value!=0)
            printf("\n Error in enabling the LEON3 uart"); // This is displayed in the jtga
debug mode

        // Set the interrupt handler and enable the interrupt
```

```
        catch_interrupt(irqhandler_rx, INTERRUPT_ID_UART1);
        enable_irq(INTERRUPT_ID_UART1);

        value = APB_UART_GET_STATUS(BASE_ADRR_UART1);
            printf("UART status reg is =%x \n ", value);

        value=UART_SEND_BYTE_RAW(BASE_ADRR_UART1,0x32);
        if (value!=0)
            printf("\n Error in sending. error=%d", value);
        value=UART_SEND_BYTE_RAW(BASE_ADRR_UART1,0x33);
        value=UART_SEND_BYTE_RAW(BASE_ADRR_UART1,0x34);
        imse_Mat SerialPicture_BGR;

        //////// Simple vga test
        printf("\n Step 1 \n");
        disable_vga(0x80000600);
        printf("\n Step 2 \n");
        init_vga(0x80000600, 1, img_dynamic, 18); // 8, with imse_setup .17
        printf("\n ENd VGA initializacion \n");
        //////

        while(is_end==0)
            {
                if (start_detection)
                {
                    //!!!!! Important Note : the SerialPicture has RGB format while the
imse_detectAndDraw works with BGR format
                    Conversion_from_RGB_to_BGR(SerialPicture, &SerialPicture_BGR);
                    SerialPicture_BGR.type=imse_MatType; //!!! We must setup the type for the
imse_8UC3
                    printf("Dirrection of prepared_vga_img is %x before \n  ",
prepared_vga_img);

                    prepare_picture_for_svga(&prepared_vga_img,SerialPicture_BGR);
                    printf("Dirrection of prepared_vga_img is %x after \n ",
prepared_vga_img);
                    set_vga_frame_addr(0x80000600, prepared_vga_img);
                    //ScaleImage_SW( SerialPicture_BGR, 720, 1280 );
                    //printf("LEON3 detection ended \n");

                    start_detection = 0;
                    printf("%c",end_of_detection); // Signalizes the end of detection
                    imse_Free(&prepared_vga_img);
                }
            }
        disable_irq(10);

        printf("\n IMSE END OF EXECUTION "); // This is displayed in the jtga debug mode
        imse_Free(&received_image.data);
        imse_Free(&SerialPicture_BGR.data_ptr);

        // vga controller
        disable_vga(0x80000600);

    #else

        int width = 80;
        int height = 60;
        int i = 0, j = 0, k = 0;

        imse_Mat *color_image_RGB = 0;
```

```
        imse_Mat *gray_image = 0;
        imse_Mat *scale_image_SW = 0;
        imse_Mat *scale_image_HW = 0;

        char c = 0;

        disable_vga (0x80000600);

        color_image_RGB = imse_CreateMat (display_height, display_width, imse_8UC3);
        gray_image = imse_CreateMat (display_height, display_width, imse_8UC1);

        for (i = 0; i < display_width * display_height * 3; i++)
        {
            color_image_RGB->data_ptr [i] = src_data [i];
        }

        init_vga (0x80000600, 1, img_dynamic, 8);   // 800x600 60 Hz

        printf ("Pulsar tecla para visualizar imagen color.\n");
        scanf ("%c", &c);

        prepare_color_image_for_svga (&prepared_vga_img, color_image_RGB);
        set_vga_frame_addr_wait_end_frame (0x80000600, prepared_vga_img);
        imse_Free (&prepared_vga_img);

        imse_CvtColor (color_image_RGB, gray_image);

        printf ("Pulsar tecla para visualizar imagen gris.\n");
        scanf ("%c", &c);

        prepare_gray_image_for_svga (&prepared_vga_img, gray_image);
        set_vga_frame_addr_wait_end_frame (0x80000600, prepared_vga_img);
        imse_Free (&prepared_vga_img);

        printf ("Pulsar tecla para comenzar el escalado en SW.\n");
        scanf ("%c", &c);

        prepared_vga_img = (unsigned int*) imse_Malloc (display_width * display_height *
sizeof (unsigned int));

        while (width <= display_width && height <= display_height)
        {
            scale_image_SW = imse_CreateMat (height, width, imse_8UC1);
            New_ScaleImage_SW (gray_image, scale_image_SW);
            fill_gray_for_svga (prepared_vga_img, scale_image_SW, display_height,
display_width);
            set_vga_frame_addr_wait_end_frame (0x80000600, prepared_vga_img);
            width = width + 80;
            height = height + 60;
            imse_Free (&scale_image_SW);
        }
        printf ("%g miliseconds in SW.\n", total_time * 1000.0);

        width = 80;
        height = 60;

        c = 0;
        total_time = 0;

        printf ("Pulsar tecla para comenzar el escalado en HW.\n");
        scanf ("%c", &c);
```

```
        while (width <= display_width && height <= display_height)
        {
            scale_image_HW = imse_CreateMat (height, width, imse_8UC1);
            New_ScaleImage_HW (gray_image, scale_image_HW);
            fill_gray_for_svga (prepared_vga_img, scale_image_HW, display_height,
display_width);
            set_vga_frame_addr_wait_end_frame (0x80000600, prepared_vga_img);
            width = width + 80;
            height = height + 60;
            imse_Free (&scale_image_HW);
        }
        printf ("%g miliseconds in HW.\n", total_time * 1000.0);

        c = 0;
        total_time = 0;

        printf ("Pulsar tecla para escalado controlado en gris.\n");
        scanf ("%c", &c);

        c = 0;

        while (c != 's')
        {
            if (c == 'e')
            {
                printf ("Introduzca ancho de imagen: ");
                scanf ("%d", &width);
                printf("%d\n", width);
                printf ("Introduzca alto de imagen: ");
                scanf ("%d", &height);
                printf("%d\n", height);

                scale_image_SW = imse_CreateMat (height, width, imse_8UC1);
                scale_image_HW = imse_CreateMat (height, width, imse_8UC1);

                New_ScaleImage_HW (gray_image, scale_image_HW);

                New_ScaleImage_SW (gray_image, scale_image_SW);

                for (i = 0; i < width * height; i ++)
                {
                    if (scale_image_SW->data_ptr [i] != scale_image_HW->data_ptr [i])
                    {
                        if (j < 10)
                        {
                            printf ("Pixel %d erroneo (SW = %d, HW = %d)\n", i,
                                        scale_image_SW->data_ptr [i],
scale_image_HW->data_ptr [i]);
                        }
                        j++;
                    }
                }


                if (j != 0)
                    printf ("Las imagenes no coinciden en %d pixels.\n", j);

                fill_gray_for_svga (prepared_vga_img, scale_image_HW, display_height,
display_width);
                set_vga_frame_addr_wait_end_frame (0x80000600, prepared_vga_img);
```

```
            c = 0;
            j = 0;
        }
        else if (c == 's')
            break;

        else
            printf ("Pulsar 's' para terminar o 'e' para realizar escalado.\n");

        scanf ("%c", &c);

        imse_Free (&scale_image_SW);
        imse_Free (&scale_image_HW);
    }

    imse_Free (&color_image_RGB);
    imse_Free (&gray_image);

    printf ("Pulsar tecla para escalado controlado en color (HW).\n");
    scanf ("%c", &c);

    c = 0;

    while (c != 's')
    {
        if (c == 'e')
        {
            printf ("Introduzca ancho de imagen: ");
            scanf ("%d", &width);
            printf("%d\n", width);
            printf ("Introduzca alto de imagen: ");
            scanf ("%d", &height);
            printf("%d\n", height);

            color_image_RGB = imse_CreateMat (height, width, imse_8UC3);
            gray_image = imse_CreateMat (display_height, display_width, imse_8UC1);
            scale_image_HW = imse_CreateMat (height, width, imse_8UC1);

            for (j = 0; j < 3; j++)
            {
                k = 0;
                for (i = 0; i < display_width * display_height * 3-3; i = i + 3, k++)
                    gray_image->data_ptr [k] = src_data [i + j];

                New_ScaleImage_HW (gray_image, scale_image_HW);

                k = 0;
                for (i = 0; i < height * width * 3-3; i = i + 3, k++)
                    color_image_RGB->data_ptr [i + j] = scale_image_HW->data_ptr [k];
            }

            fill_color_for_svga (prepared_vga_img, color_image_RGB, display_height,
display_width);
            set_vga_frame_addr_wait_end_frame (0x80000600, prepared_vga_img);

            c = 0;
        }

        else if (c == 's')
            break;
```

```
            else
                printf ("Pulsar 's' para terminar programa o 'e' para realizar escalado.\n");

            scanf ("%c", &c);

            imse_Free (&color_image_RGB);
            imse_Free (&gray_image);
            imse_Free (&scale_image_HW);
        }

        imse_Free (&color_image_RGB);
        imse_Free (&gray_image);
        imse_Free (&scale_image_SW);
        imse_Free (&scale_image_HW);
        imse_Free (&prepared_vga_img);
        free (img_dynamic);

        disable_vga (0x80000600);
    #endif

    return 0;
}


#if LEON3_UART_CONTROLLED
// Ported OpenCV implementation. It uses integer operations
static void ScaleImage_SW( imse_Mat raw_frame, int height, int width )
{
   imse_Mat *imse_gray = 0;
   imse_Mat *imse_scale_SW = 0;
   imse_Mat *imse_scale_HW = 0;

   int i, j = 0;
   clock_t t_ini, t_fin;
   double secs;

   imse_gray = imse_CreateMat( raw_frame.rows, raw_frame.cols, imse_8UC1 );
   imse_scale_SW = imse_CreateMat( height, width, imse_8UC1 );
   imse_scale_HW = imse_CreateMat( height, width, imse_8UC1 );

   imse_CvtColor(&raw_frame, imse_gray);

   for (i = 0; i < imse_gray->rows * imse_gray->cols; i++)
       memory_data [i] = imse_gray->data_ptr [i];

   for (i = 0; i < 10; i++)
       printf ("data_%d = %d\n", i, memory_data [i]);

   t_ini = clock ();
   imse_Resize(imse_gray, imse_scale_SW, imse_INTER_LINEAR);
   t_fin = clock ();

   secs = (double) (t_fin - t_ini) / CLOCKS_PER_SEC;
   printf ("%.3g miliseconds.\n", secs * 1000.0);

   ScaleImage_HW( imse_gray, imse_scale_HW, width, height );

   for (i = 0; i < height * width; i++)
   {
       if (imse_scale_SW->data_ptr[i] == data_out [i])
```

```
        {
        }
        else
        {
            if (j < 3)
            {
                printf ("Las imagenes no coincien en pixel %d. Model = %d and HW = %d\n", i,
                                imse_scale_SW->data_ptr [i], data_out [i]);
            }
            j++;
        }
    }

    if (j == 0)
        printf ("Escalado finalizado con exito.\n");
    else
        printf ("El numero de fallos es de %d\n", j);

    imse_ReleaseMat( &imse_gray);
    imse_ReleaseMat( &imse_scale_SW);
    imse_ReleaseMat( &imse_scale_HW);

    wait_for_interrupt = 1; // global variable that is changed into interrupt handling routine
to zero
    enable_irq(10);
}

static void ScaleImage_HW( imse_Mat* src_image, imse_Mat* dst_image, int width, int height)
{
    const int INTER_RESIZE_COEF_BITS = 11;
    const int IMSE_EXTRA_PRECISION_BITS = 4;
    const int IMSE_RESIZE_COEF_SCALE = 1 << (INTER_RESIZE_COEF_BITS +
IMSE_EXTRA_PRECISION_BITS);

    int *TFG_BASE_ADDR = (int *) 0x80000A00;

    clock_t t_ini, t_fin;
    double secs;

    int shifted_scale_alpha = (IMSE_RESIZE_COEF_SCALE * src_image->cols) / width;
    int shifted_scale_beta = (IMSE_RESIZE_COEF_SCALE * src_image->rows) / height;

    int i;

    int aux = 1, int_address_width = 0, src_address_width = 0, dst_address_width = 0,
address_height = 0;
        char flag1 = 0, flag2 = 0, flag3 = 0, flag4 = 0;

        for (i = 0; i < 12; i++)
        {
            aux = aux << i;

            if (aux > src_image->cols && flag1 == 0)
            {
                int_address_width = i;
                flag1 = 1;
            }
            if (aux > (src_image->cols / 4) && flag2 == 0)
            {
                src_address_width = i;
                flag2 = 1;
```

```
            }
            if (aux > (width / 4) && flag3 == 0)
            {
                dst_address_width = i;
                flag3 = 1;
            }
            if (aux > src_image->rows && flag4 == 0)
            {
                address_height = i;
                flag4 = 1;
            }

        aux = 1;
    }

    TFG_BASE_ADDR [1] = 0;                          // reset for new transfer
    TFG_BASE_ADDR [2] = (int) &memory_data; //(int) src_image->data_ptr;    // read address
    TFG_BASE_ADDR [3] = (int) &data_out;    //(int) dst_image->data_ptr;    // write address
    TFG_BASE_ADDR [4] = src_image->cols;            // source row size
    TFG_BASE_ADDR [5] = src_image->rows;            // source col size
    TFG_BASE_ADDR [6] = width;          // destination row size
    TFG_BASE_ADDR [7] = height;         // destination col size
    TFG_BASE_ADDR [8] = shifted_scale_alpha;        // row scale coefficient
    TFG_BASE_ADDR [9] = shifted_scale_beta;         // col scale coefficient
    TFG_BASE_ADDR [1] = 0x1;                        // start and read

    t_ini = clock ();

    while (TFG_BASE_ADDR [0] == 1)
        {
        }

    t_fin = clock ();

    secs = (double) (t_fin - t_ini) / CLOCKS_PER_SEC;
    printf ("%.3g miliseconds.\n", secs * 1000.0);

    printf ("src_data: %x\n", TFG_BASE_ADDR [2]);
    printf ("dst_data: %x\n", TFG_BASE_ADDR [3]);
    printf ("src_width: %d\n", TFG_BASE_ADDR [4]);
    printf ("src_height: %d\n", TFG_BASE_ADDR [5]);
    printf ("dst_width: %d\n", TFG_BASE_ADDR [6]);
    printf ("dst_height: %d\n", TFG_BASE_ADDR [7]);
    printf ("shifted_scale_alpha: %d\n", TFG_BASE_ADDR [8]);
    printf ("shifted_scale_beta: %d\n", TFG_BASE_ADDR [9]);

    printf ("int_address_width: %d\n", int_address_width);
    printf ("src_address_width: %d\n", src_address_width);
    printf ("dst_address_width: %d\n", dst_address_width);
    printf ("address_height: %d\n", address_height);
}

#else

int prepare_gray_image_for_svga (unsigned int** dst_image, imse_Mat* src_img)
{
    unsigned int *fbase = 0;
    unsigned int i = 0, col, row, index;
    unsigned char gray_pixel;

    *dst_image = (unsigned int*) imse_Malloc (src_img->cols * src_img->rows * sizeof (unsigned
```

```
int));
    if (*dst_image == NULL)
    {
        fail(1);
        return 1;
    }

    fbase = *dst_image;
    for(row = 0; row < (unsigned int) src_img->rows; row++)
    {
        for(col = 0; col < (unsigned int) src_img->cols; col++)
        {
            index = row * src_img->cols + col;
            gray_pixel = *(src_img->data_ptr + i);
            *(fbase + i) = gray_pixel + (gray_pixel << 8) + (gray_pixel << 16);
            i++;
        }
    }

    return 0;
}

int fill_gray_for_svga (unsigned int* buffer, imse_Mat* src_img, int buffer_rows, int
buffer_cols)
{
    unsigned int col, row, index_buf, index_src, rows_res, cols_res;
    unsigned char gray_pixel;
    index_buf = 0;
    index_src = 0;

    if ((src_img->rows > buffer_rows) || (src_img->cols > buffer_cols))
    {
        for (row = 0; row < (unsigned int) buffer_rows; row ++)
        {
            for (col = 0; col < (unsigned int) buffer_cols; col++)
            {
                gray_pixel = *(src_img->data_ptr + index_src);
                *(buffer + index_buf) = gray_pixel + (gray_pixel << 8) + (gray_pixel << 16);
                index_buf++;
                index_src++;
            }
            index_src = row * src_img->cols;
        }
    }
    else
    {
        rows_res = src_img->rows % 2;
        cols_res = src_img->cols % 2;

        for (row = 0; row < (unsigned int) ((buffer_rows / 2) - ((src_img->rows / 2) -
rows_res)); row ++)
        {
            for (col = 0; col < (unsigned int) buffer_cols; col++)
            {
                *(buffer + index_buf) = 0x0 + (0x0 << 8) + (0x0 << 16); // resto de pixeles en
negro
                index_buf++;
            }
        }
        for (row = ((buffer_rows / 2) - ((src_img->rows / 2) - rows_res)); row < (unsigned
int)
```

```
                                                ((buffer_rows / 2) + (src_img->rows / 2)); row ++)
        {
            for (col = 0; col < (unsigned int) ((buffer_cols / 2) - (src_img->cols / 2)); col+
+)
            {
                *(buffer + index_buf) = 0x0 + (0x0 << 8) + (0x0 << 16); // resto de pixeles en
negro
                index_buf++;
            }
            for(col = ((buffer_cols / 2) - (src_img->cols / 2)); col < (unsigned int)
((buffer_cols / 2) +
                                                                ((src_img->cols / 2) + cols_res));
col++)
            {
                gray_pixel = *(src_img->data_ptr + index_src);
                *(buffer + index_buf) = gray_pixel + (gray_pixel << 8) + (gray_pixel << 16);
                index_buf++;
                index_src++;
            }
            for (col = ((buffer_cols / 2) + ((src_img->cols / 2) + cols_res)); col < (unsigned
int)
                                                                                buffer_cols;
col++)
            {
                *(buffer + index_buf) = 0x0 + (0x0 << 8) + (0x0 << 16); // resto de pixeles en
negro
                index_buf++;
            }
        }
        for (row = ((buffer_rows / 2) + (src_img->rows / 2)); row < (unsigned int)
buffer_rows; row ++)
        {
            for (col = 0; col < (unsigned int) buffer_cols; col++)
            {
                *(buffer + index_buf) = 0x0 + (0x0 << 8) + (0x0 << 16); // resto de pixeles en
negro
                index_buf++;
            }
        }
    }
    return 0;
}

int prepare_color_image_for_svga (unsigned int** dst_image, imse_Mat* src_img)
{
    unsigned int *fbase = 0;

    unsigned int i = 0, col, row, index;

    *dst_image = (unsigned int*) imse_Malloc (src_img->cols * src_img->rows*sizeof (unsigned
int));
    if (*dst_image == NULL)
    {
        fail(1);
        return 1;
    }

    fbase = *dst_image;
    // copy the content of the BGR pixels into RGB format
    for(row = 0; row < (unsigned int) src_img->rows; row++)
    {
```

```
        for(col = 0; col < (unsigned int) src_img->cols; col++)
        {
            index = row * src_img->cols + col;
            *(fbase+i) = (*(src_img->data_ptr + 3 * index + 0)) + ((*(src_img->data_ptr + 3 *
index + 1)) << 8)
                                                                + ((*(src_img->data_ptr + 3 *
index + 2)) << 16);
            i++;
        }
    }
    return 0;
}

int fill_color_for_svga (unsigned int* buffer, imse_Mat* src_img, int buffer_rows, int
buffer_cols)
{
    unsigned int col, row, index_buf, index_src, rows_res, cols_res;
    index_buf = 0;
    index_src = 0;

    if ((src_img->rows > buffer_rows) || (src_img->cols > buffer_cols))
    {
        for (row = 0; row < (unsigned int) buffer_rows; row ++)
        {
            for (col = 0; col < (unsigned int) buffer_cols; col++)
            {
                *(buffer + index_buf) = *(src_img->data_ptr + index_src) +
                (*(src_img->data_ptr + index_src + 1) << 8) +
                (*(src_img->data_ptr + index_src + 2) << 16);
                index_buf++;
                index_src = index_src + 3;
            }
            index_src = row * src_img->cols * 3;
        }
    }
    else
    {
        rows_res = src_img->rows % 2;
        cols_res = src_img->cols % 2;

        for (row = 0; row < (unsigned int) ((buffer_rows / 2) - ((src_img->rows / 2) -
rows_res)); row ++)
        {
            for (col = 0; col < (unsigned int) buffer_cols; col++)
            {
                *(buffer + index_buf) = 0x0 + (0x0 << 8) + (0x0 << 16); // resto de pixeles en
negro
                index_buf++;
            }
        }
        for (row = ((buffer_rows / 2) - ((src_img->rows / 2) - rows_res)); row < (unsigned
int)
                                            ((buffer_rows / 2) + (src_img->rows / 2)); row ++)
        {
            for (col = 0; col < (unsigned int) ((buffer_cols / 2) - (src_img->cols / 2)); col+
+)
            {
                *(buffer + index_buf) = 0x0 + (0x0 << 8) + (0x0 << 16); // resto de pixeles en
negro
                index_buf++;
            }
```

```
            for(col = ((buffer_cols / 2) - (src_img->cols / 2)); col < (unsigned int)
((buffer_cols / 2) +
                                                                ((src_img->cols / 2) + cols_res));
col++)
            {
                *(buffer + index_buf) = *(src_img->data_ptr + index_src) +
                (*(src_img->data_ptr + index_src + 1) << 8) +
                (*(src_img->data_ptr + index_src + 2) << 16);
                index_buf++;
                index_src = index_src + 3;
            }
            for (col = ((buffer_cols / 2) + ((src_img->cols / 2) + cols_res)); col < (unsigned
int)
                                                                                buffer_cols; col+
+)
            {
                *(buffer + index_buf) = 0x0 + (0x0 << 8) + (0x0 << 16); // resto de pixeles en
negro
                index_buf++;
            }
        }
        for (row = ((buffer_rows / 2) + (src_img->rows / 2)); row < (unsigned int)
buffer_rows; row ++)
        {
            for (col = 0; col < (unsigned int) buffer_cols; col++)
            {
                *(buffer + index_buf) = 0x0 + (0x0 << 8) + (0x0 << 16); // resto de pixeles en
negro
                index_buf++;
            }
        }
    }
    return 0;
}

static void New_ScaleImage_SW (imse_Mat* src_image, imse_Mat* dst_image)
{
    clock_t t_ini, t_fin;

    t_ini = clock ();
    imse_Resize(src_image, dst_image, imse_INTER_LINEAR);
    t_fin = clock ();

    secs = (double) (t_fin - t_ini) / CLOCKS_PER_SEC;
    printf ("%g miliseconds in SW for an image of size %dx%d.\n", secs * 1000.0,
dst_image->cols, dst_image->rows);
    total_time = total_time + secs;
}

static void New_ScaleImage_HW (imse_Mat* src_image, imse_Mat* dst_image)
{
    const int INTER_RESIZE_COEF_BITS = 11;
    const int IMSE_EXTRA_PRECISION_BITS = 4;
    const int IMSE_RESIZE_COEF_SCALE = 1 << (INTER_RESIZE_COEF_BITS +
IMSE_EXTRA_PRECISION_BITS);

    int *TFG_BASE_ADDR = (int *) 0x80000A00;

    clock_t t_ini, t_fin;

    int shifted_scale_alpha = (IMSE_RESIZE_COEF_SCALE * src_image->cols) / dst_image->cols;
```

```
    int shifted_scale_beta = (IMSE_RESIZE_COEF_SCALE * src_image->rows) / dst_image->rows;

    TFG_BASE_ADDR [1] = 0;                          // reset for new transfer
    TFG_BASE_ADDR [2] = (int) src_image->data_ptr;  // read address
    TFG_BASE_ADDR [3] = (int) dst_image->data_ptr;  // write address
    TFG_BASE_ADDR [4] = src_image->cols;            // source row size
    TFG_BASE_ADDR [5] = src_image->rows;            // source col size
    TFG_BASE_ADDR [6] = dst_image->cols;            // destination row size
    TFG_BASE_ADDR [7] = dst_image->rows;            // destination col size
    TFG_BASE_ADDR [8] = shifted_scale_alpha;        // row scale coefficient
    TFG_BASE_ADDR [9] = shifted_scale_beta;         // col scale coefficient

    t_ini = clock ();

    TFG_BASE_ADDR [1] = 0x1;                        // start and read

    while (TFG_BASE_ADDR [0] == 1);

    t_fin = clock ();

    secs = (double) (t_fin - t_ini) / CLOCKS_PER_SEC;
    printf ("%g miliseconds in HW for an image of size %dx%d.\n", secs * 1000.0,
dst_image->cols, dst_image->rows);
    total_time = total_time + secs;
}
#endif

// video decoding
#if DISPLAY_PICTURE

/*
 * svgactrl_test(...)
 *
 * Arguments:
 *
 * addr      - Address of core registers
 * real      - See description at top of file
 * alloc     - Allocate memory dynamically, otherwise fbaddress is used
 * fbaddress - Frame buffer address
 * format    - Selects format or -1 for autodetect
 * delay     - Counter that delays core disable, -1 to delay forever
 * blank     - Blank screen before core enable
 *
 */
int svgactrl_test(unsigned int addr, unsigned int real, unsigned int alloc,
                  unsigned int fbaddress, int format, int delay, int blank)
{
  unsigned int i, x ;// y, vec[3];
  int found = 0;
  int clock;
  unsigned int *fbase;

  format_s *f;

  struct svgactrlregs *regs;

    printf("...SVAGA: PASS 1 \n");


  regs = (struct svgactrlregs*)(addr);
```

```
regs->stat = 0;



 printf("...SVAGA: PASS 2 \n");
if (real) {
   if (format == -1) {
      /* Scan available clocks */
      for (format = 0; format < NUM_FORMATS && !found; format++) {
         for (clock = 0; clock < 4; clock++) {
            if (formats[format].period == regs->dclock[clock]) {
               found = 1;
               break;
            }
         }
      }
   } else {
      /* Check if clock for format is available */
      for (clock = 0; clock < 4; clock++) {
         if (formats[format].period == regs->dclock[clock]) {
            found = 1;
            break;
         }
      }
   }
} else {
   /* Must have a clock with period != 0 */
   format = TEST_FORMAT;
   for (clock = 0; clock < 4; clock++) {
      if (regs->dclock[clock]) {
         found = 1;
         break;
      }
   }
}

if (!found) {
   fail(0);
}

  printf("...SVAGA: PASS 3 \n");



f = &formats[format];

regs->vidlen = ((f->vactive_video-1) << 16)  | (f->hactive_video-1);
regs->fporch = ((f->vfporch) << 16)  | (f->hfporch);
regs->synclen = ((f->vsync) << 16)  | (f->hsync);
regs->linelen = (((f->vactive_video-1 + f->vfporch + f->vsync + f->vbporch) << 16) |
                 (f->hactive_video-1 + f->hfporch + f->hsync + f->hbporch));

if (alloc) {
   /* Allocate framebuffer */

  fbase = (unsigned int*)imse_Malloc(2*f->hactive_video*f->vactive_video);
    if (fbase == NULL) {
      fail(1);
   }
} else {
   /* Use specific frame buffer base address */
```

```
      fbase = (unsigned int*)fbaddress;
   }
   regs->framebuf = (int)fbase;

   if (blank) {
      /* Blank screen */
      for (x = 0; x < f->hactive_video*f->vactive_video/2; x++) {
         fbase[x] = 0x00000000;
      }
   }

   /* Enable controller */

   regs->stat = ((clock << SVGACTRL_CLKSEL) | (3 << SVGACTRL_BDSEL) |  //32 bit mode
                 (1 << SVGACTRL_EN));

printf("...SVAGA: PASS 4 \n");
   if (delay != -1) {
      for (i = 0; i < (unsigned int)delay; i++)
         ;
   } else {
      while(1)
         ;
   }

   /* Deactivate core */
   regs->stat = 0;
printf("...SVAGA: PASS 5 \n");


   return 0;
}


int prepare_picture_for_svga(unsigned int** video_address, imse_Mat img)
{
    unsigned int *fbase= 0;

    unsigned int i, col, row, index;
     /* Allocate framebuffer */

    *video_address = (unsigned int*)imse_Malloc(img.cols*img.rows*sizeof(unsigned int));
    if (*video_address == NULL)
    {
            fail(1);
        return 1;
    }

    fbase = *video_address;
    // copy the content of the BGR pixels into RGB format
    for(row=0; row<(unsigned int) img.rows; row++)
    {
      for(col=0; col<(unsigned int) img.cols; col++)
      {
          index=row*img.cols+col;
          *(fbase+i)= (*(img.data_ptr+3*index+0))+((*(img.data_ptr+3*index+1))<<8)+((*
(img.data_ptr+3*index+2))<<16);
           i++; //copy picture.
      }

      /*
```

```
     for(index=i; index<(i+800-img.cols); index++)
     {
       *(fbase+index)=255<<16;// Takes the Red color for one hortizontal line
     }
      i=index;
    }

    // red until the end
    for(index=i; index<800*640; index++)
     {
       *(fbase+index)=255<<16;// Takes the Red color
     }

  */
    }
return 0;
}


#endif
```