ISSN 2007-9737

HW/SW Co-Design of a Specific Accelerator for Robotic Computer Vision

Adrian Pedroza de la Crúz[1], Miguel Ángel Carrazco Díaz[1], Susana Ortega Cisneros[1], Juan José Raygoza Panduro[2], Jorge Rivera Domínguez[3], Federico Sandoval Ibarra[1]

[1] Instituto Politécnico Nacional, CINVESTAV Jalisco, Mexico

[2] University of Guadalajara, CUCEI, Jalisco, Mexico

[3] Instituto Politécnico Nacional, commissioned as a CONACyT professorship to CINVESTAV, Jalisco, Mexico

{apedroza, mcarrazco, sortega, riveraj, sandoval}@gdl.cinvestav.mx, juan.raygoza@cucei.udg.mx

Abstract. This paper presents an image processing application focused on robotic computer vision. The co-design is divided into three main parts: a hardware accelerator, a PCIe® based framework for HW/SW link, and application software. The implemented accelerator performs preprocessing for facial recognition in order to reduce the workload in the main system processor. The hardware layer is implemented in Altera FPGAs, while the project software layer provides a device driver for Linux to link the user application with the coprocessor. The user application controls the data transfer between the operating system and the device driver. The platform allows rapid prototyping of accelerators, taking advantage of the duality of a programmable hardware and a general purpose processor connected through a PCIe® link. The proposed architecture enables co-design of various image processing algorithms. In this case, the results of the design of an accelerator that performs histogram equalization for contrast correction of color images are presented.

Keywords. Accelerator for computer vision, design automation, field-programmable gate array (FPGA), hardware accelerator, hardware design, high performance computing, Linux driver, PCIe framework, Verilog.

1 Introduction

For facial and object recognition [1], it is important to obtain a clear picture to be processed. A technique for correcting overexposure or underexposure is histogram equalization as preprocessing in order to obtain a sharper image. As this is a demanding task for the processor, it is recommended that this be performed via hardware [2].

For the implementation of processing algorithms in HW/SW, when there are no optimal development tools, much time is required for debugging the design.

One way to develop accelerators is to have a reusable framework [3, 4] in order to reduce design time. Fig. 1 shows the HW/SW framework components.

The aim of the PCIe®-FPGA framework proposed in this paper is communication between the software (User Application) and the hardware accelerator configured as an end point [5, 6].

The architecture is composed of the following elements: a PCIe® communication link, internal memory access, a configuration slave module, a read master module, and a write master module. All these modules are connected to the Avalon® MM Altera®'s interface [7].

In order to modify the accelerator functionality to be performed, a tool that facilitates the communication and tests hardware accelerators is used employing the same PCIe® driver. The design should be accomplished with a simple

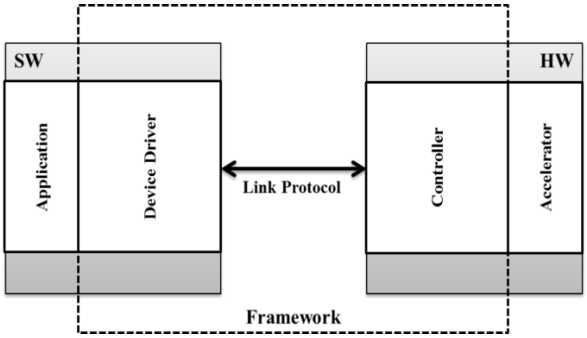

Fig. 1. Standard framework for accelerator development

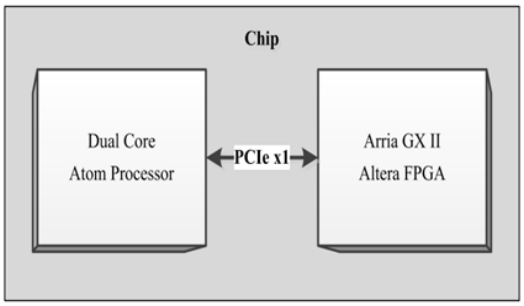

Fig. 2. Reconfigurable processor

interface in order to interact with the slave module and the master modules.

2 Framework

The platform contains a reconfigurable processor, so called because the package itself contains an Intel® Atom™ Dual-Core processor [8] and an Altera® Arria® GX II FPGA [9]. Both are interconnected by a PCIe® port. The FPGA can be used as a coprocessor or an accelerator. It can run specific high performance functions, while a general purpose processor would need much more time in order to accomplish the same task.

Fig. 2 shows a block representation of the components interconnected in the reconfigurable processor [10].

2.1 Hardware Framework

The framework is a HW/SW co-design [11]. On the side of the hardware architecture, a PCIe® communication control is required, where the data, the operation, and the control signals are arbitrated in order to avoid collisions and ensure the correct functionality of the system. The control signals indicate the accelerator functionality.

The implemented architecture is shown in Fig. 3, where all framework components and the accelerator interconnection will be described in the subsections that follow.

a. PCIe® IP Block

The communication with the accelerator is performed by the protocol PCIe. The Arria® GX II FPGA includes an integrated hard IP block which implements the PCIe physical interface, the data link, and the transaction layer. This block is connected to the root complex on the outside and to the Avalon® MM bridge on the inside.

b. Avalon® MM Bridge

The function of this block is to convert PCI Express packets to Avalon® Memory Mapped (MM) transactions and vice versa.

c. Avalon® Bus

The Avalon® bus is the interconnected system that communicates the Avalon® MM bridge, the on-chip memory, the master reading interface, the master writing interface, and the slave configuration interface.

d. On-Chip Memory

The purpose of the on-chip memory is to store all received data from the reading port to be processed and to save the resulting data while the port waits to transmit the data back. The memory used is dual port in order to enable reading and writing simultaneously.

e. Read Master

The read master is responsible for controlling the memory address and the data transfer toward the

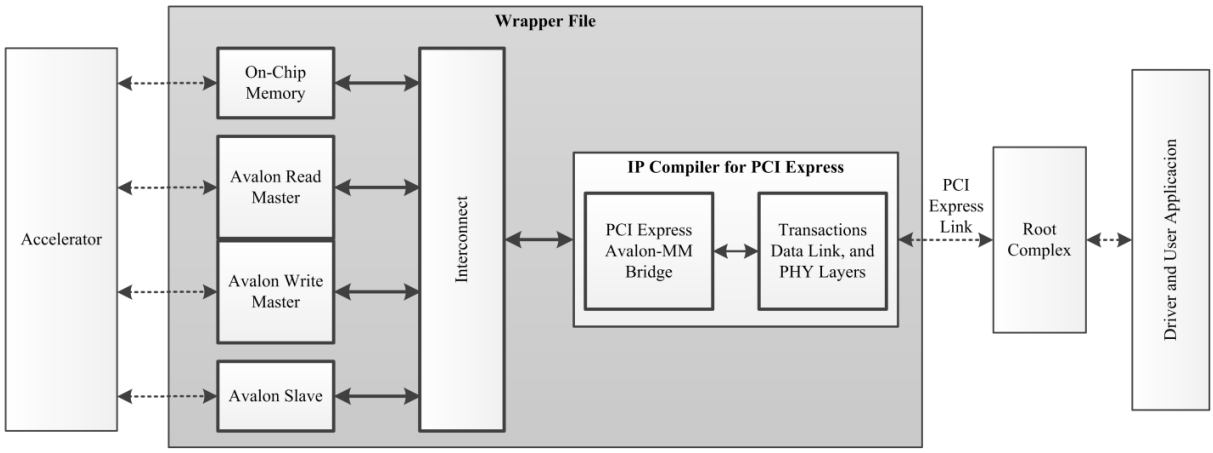

Fig. 3. Schematic representation of hardware architecture implementation

accelerator, ensuring the proper transfer time and the required organization.

f. Write Master

The write master is responsible for receiving data from the accelerator. In order to provide the format required for storing in memory, it must monitor bus saturation to pause and resume the accelerator activity.

g. Configuration Slave

The accelerator needs configuration instructions such as start and stop to be operative. These commands are transferred from the bus to the accelerator by the configuration slave. Following the data path, when the software application uses hardware acceleration, it begins to request a communication channel. When the channel is obtained, the configuration parameters are sent, and then the data packets are transferred from the host to the PCIe® physical layer in the FPGA, where the received data package is transferred to the Avalon® MM bridge. Once data enters the system bus, the control instructions are sent to the slave configuration, where the processed data is stored in memory. After the accelerator is properly configured and started, the read master receives the data from the memory and it is sent to be processed by the accelerator core. In a few clock cycles, the resulting data is obtained, the write master is activated, and it is responsible for returning the data results to the memory. When the resulting data are ready, the software application requests them to the Avalon® bus and it retrieves all data from memory that delivers it to the system bus. Subsequently, the data is transferred from the bus to the Avalon® MM bridge, and finally the data packages are transferred from the bridge to the physical layer in order to reach the main host.

2.2 Software Framework

The PCIe® Linux driver [12] is compiled on a Timesys 14 operating system (OS), which is based on Fedora distribution, with a 2.635-9 Kernel. This OS contains predefined basic configurations for optimal platform functionality. The driver implements methods for hardware configuration of read and write communication via PCIe®. The user application is coded in C and controls the driver functions.

a. PCIe® Hard IP Block

The Arria® GX II hard IP block for PCIe® is able to use x1, x4, or x8 lines, but the physical interconnection with Atom™ processor is x1. The PCIe® protocol version supported by FPGA is 1.1 with a transfer rate of 2.5 gigatransfers per second and supports 6 BARs (Base Address Register) of

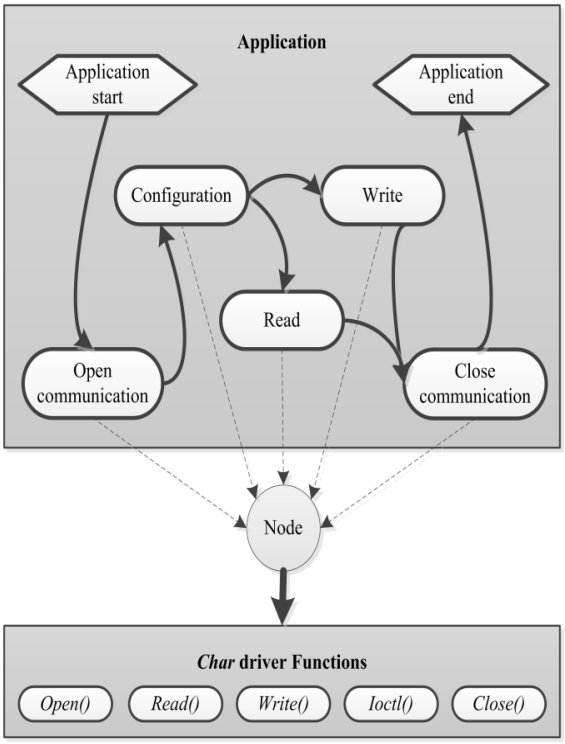

Fig. 4. Test application communication flow diagram

32 non-prefetchable bits or 64 prefetchable bits and it admits MSI (Message Signaled Interrupts) of 64 address bits.

b. Linux Driver

The driver contains the fundamental steps to be installed in the platform. It begins looking for Altera®'s vendor_id, device_id, and revision_id (0x1172, 0xE001, and 0x01, respectively). Once a device is found, it is enabled and the BAR is configured. Only BAR0 is configured in this case. Finally, connectivity and interruption tests are performed in order to ascertain optimal functionality of the device, preparing it for use.

This driver is char type, controlled by the user application coded in C, and it requires a Linux node to perform the following functions:

- Open: turns on the PCIe® device, putting the unit into use.
- Close: turns off communication with the PCIe®.
- Read: performs data read from memory to the device.
- Write: performs data write from the device to the memory.
- Ioctl: sets the peripheral device to be properly addressed.

The write function and the read function are able to transfer data memory blocks as a single data package.

2.3 Test Application

A base application has been developed to control the driver and the transfers between the device memory and the OS application. Fig. 4 shows a block diagram of communication flow between software applications and driver functions.

Transfer tests were performed from 1 Kb to the maximum memory in FPGA of 500 Kb; however, the driver can perform transfers of approximately 100 Mb with this operating system kernel configuration.

3 Hardware Accelerator

The accelerator was used to test the architecture as an image processor. This processor performs image equalization [13] of overexposed or underexposed images.

Let us consider a source color image $A = \{a(i,j) \mid 1 \le i \le M, 1 \le j \le N\} \in \mathbb{R}^{(m \times n)}$, where $a(i,j)$ is an element of the image. It is assumed that A has a dynamic range between $[l_1, l_2] \in \mathbb{R}$ where $a(i,j) \in [l_1 l_2]$ and $l_1 < l_2$ [14]. Transforming the image A to a discrete representation is expressed as three superimposed matrices $D = R, G, B$ of m by n dimension, as shown in Equation 1.

Each pixel is composed of elements in a concatenation $d_{ij} = \{r_{ij}, g_{ij}, b_{ij}\} \in \mathbb{Z}^+$, where each element of a pixel is represented by a binary width word x, where the maximum number of

Table 1. Base address registers

Address	Framework Component	Description
0x00000000 - 0x0001FFFF	On-chip memory	FPGA Memory
0x00020000	Slave Accelerator	Source Image Address
0x00020001	Slave Accelerator	Target Image Address
0x00020002	Slave Accelerator	Image Length
0x00020003	Slave Accelerator	Pixel Height
x00020004	Slave Accelerator	Pixel Width
0x00020005	Slave Accelerator	Command
0x00020006	Slave Accelerator	Start bit
0x00020007	Slave Accelerator	Process Time (Read Only)
0x00020008	Slave Accelerator	Status (Read Only)
0x00020009 - 0x0002000F	Slave Accelerator	Unused

combinations is $W = 2^x$ within the range $l \leq d_{ij} \leq u \in \mathbb{Z}^+$, where $l = 0, u = 2^x - 1 \in \mathbb{Z}^+$.

$$D = \begin{bmatrix} d_{11} & d_{12} & \cdots & d_{1m} \\ d_{21} & d_{22} & \cdots & d_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n1} & d_{n2} & \cdots & d_{nm} \end{bmatrix}. \quad (1)$$

Obtaining the resulting image requires applying a correction algorithm to the source image by replacing each one of the elements in the matrices R, G, B with the values obtained from the transformation vector T, where $f(a(i, j)) = \{t_{rij}, t_{gij}, t_{bij}\}$. The vector T is obtained by calculating the steps described in what follows.

3.1 Function of Probability

The probability of the pixel value is determined by the relationship $P(h) = v_h / v$, where v is the number of possible combinations and v_h is the number of times the element value corresponds to *h*. The possible values are within the range $L \leq v_h \leq U \in \mathbb{Z}^+$ with a lower limit of $L = 0$ and an upper limit of $U = v - 1$.

3.2 Distribution Function

The Histogram H [15] is the representation of the probability distribution of data values, considering the amount of each different value of the elements in the matrix characterized by the following vector $H = h_0, h_1, \cdots, h_{2^x - 1}$ evaluated by summation in (2):

$$m \times n = \sum_{i=0}^{h_{2^x-1}} h_i. \quad (2)$$

The following example shows the C code for obtaining the histogram using the software.

```
// Distribution
  for (i=0; i<height; i++) {
    for (j=0; j<width; j++) {
      for (k=0; k<(bppx/8); k++) {
        den[k][buffer[(bppx/8)*(i*width+j)+k]]++;
      }
    }
  }
```

The code above is converted to Verilog code to be synthetized and use the generated hardware to obtain the histogram.

```
HIST:

begin
  if(read_valid == 1'b1)
  begin
    cnt_en_r <= {{W-1{1'b0}},{1'b1}} << wire_r;
    cnt_en_g <= {{W-1{1'b0}},{1'b1}} << wire_g;
    cnt_en_b <= {{W-1{1'b0}},{1'b1}} << wire_b;
    state    <= HIST;
  end
  else
  begin
    if(read_done == 1'b1)
    begin
      reg_full <= 1'b1;
      cnt_en_r <= {W{1'b0}};
      cnt_en_g <= {W{1'b0}};
      cnt_en_b <= {W{1'b0}};
      state    <= CUMU;
    end
    else
    begin
      state    <= HIST;
    end
  end
end
```

3.3 Accumulation Function

The cumulative distribution function C is represented by the vector $C = c_0, c_1, \cdots, c_{2^x-1}$ in which the element values are given by Equation 3 and where $c_0 = h_0$.

$$c_i = \sum_{i=1}^{c_{2^x-1}} c_{i-1} + h_i. \tag{3}$$

Adding an element to the end of the vector c_{2^x} and assuming $c_0 = 0$ result in summation (4):

$$c_i = \sum_{i=1}^{c_{2^x}} c_{i-1} + h_{i-1}. \tag{4}$$

The following lines show the C code which obtains the cumulative distribution function for the software:

```
// Cumulative
  for (i=1; i<=256; i++) {
    for (k=0; k<(bppx/8); k++)  {
      cum[k][i] = den[k][i-1] + cum[k][i-1];
    }
  }
```

Then, we give the C code by which the obtained cumulative distribution function is converted to Verilog in order to be synthesized in hardware:

```
CUMU:

begin
  if(cnt_cumu < W+4)
  begin
    if(cnt_cumu < W)
    begin
      reg_cumu_1 <=
      cnt_hist_r[wire_cumu_0[X-1:0]];
    end
    if(cnt_cumu > 0 && cnt_cumu < W+1)
    begin
      reg_cumu_2 <= reg_cumu_1 +
      cnt_hist_g[wire_cumu_1[X-1:0]];
    end
    if(cnt_cumu > 1 && cnt_cumu < W+2)
    begin
      reg_cumu_3 <= reg_cumu_2 +
     cnt_hist_b[wire_cumu_2[X-1:0]];
    end
    if(cnt_cumu > 2 && cnt_cumu < W+3)
    begin
      reg_cumu_c <= reg_cumu_3 + reg_cumu_c;
    end
    if(cnt_cumu > 3 && cnt_cumu < W+4)
    begin
      reg_cumu[wire_cumu_4[X-1:0]] <=
      reg_cumu_c;
    end
    cnt_en_c <= 1'b1;
    state <= CUMU;
  end
  else begin
    cnt_en_c <= 1'b0;
```

```
    state <= TRAN;
  end
end
```

3.4 Transformation Function

The proposed transformation function is calculated with the vector $T = t_0, t_1, \cdots, t_{2^x-1}$ using Equation 5:

$$t_i = \sum_{i=0}^{t_{2^x-1}} \frac{(2^x - 1)c_{i+1}}{m \cdot n}. \quad (5)$$

The following C code is executed by the software in order to acquire the transformation function:

```
// Transformation

  for (i=0; i<256; i++) {
    for (k=0; k<(bppx/8); k++) {
      tra[k][i] = (255*(cum[0][i+1]+cum[1][i+1]+
      cum[2][i+1]))/(height*width*3);
    }
  }
```

The transformation function is obtained through the hardware synthesis of the following Verilog code:

```
TRAN:

begin
  if(cnt_tran < W+PIPE_STEPS+3)
  begin
    if(cnt_tran < W)
    begin
      reg_shift <=
      reg_cumu[wire_tran_0[X-1:0]] << X;
    end
    if(cnt_tran > 0 && cnt_tran < W+1)
    begin
      reg_dividend <=
      reg_shift –
      reg_cumu[wire_tran_1[X-1:0]];
    end
    if(cnt_tran > 1 && cnt_tran < W+2)
    begin
      dividend <= reg_dividend;
      divisor <=
      reg_divisor[COLOR_NUM*X-1:0];
    end
    if(cnt_tran > PIPE_STEPS+2 &&
    cnt_tran < W+PIPE_STEPS+3) begin
      reg_tran[wire_tran_p[X-1:0]] <=
      quotient[X-1:0];
    end
    state   <= TRAN;
    cnt_en_t <= 1'b1;
  end
  else
  begin
    reg_full <= 1'b0;
    cnt_en_t <= 1'b0;
    state   <= SCAN;
  end
end
```

Let $O = \{o(i, j) \mid 1 \le i \le m, 1 \le j \le n\} \in \mathbb{Z}^+$ be the resulting image where it is assumed that O has a dynamic range within $[l, u] \in \mathbb{Z}^+$ and with the possible values of $o(i, j) \in [l, u]$, $l = 0, u = 2^x - 1.$

The transformation of the source image D is given to a transformation function $O = F_T(D)$ where $o(i, j) = \{t_{rij}, t_{gij}, t_{bij}\}$.

The resulting image is obtained via the software using the following C code:

```
// New Image
  for (i=0; i<height; i++) {
    for (j=0; j<width; j++) {
      for (k=0; k<(bppx/8); k++) {
        buffer[(bppx/8)*(i*width+j)+k] =
        tra[k][buffer[(bppx/8)*(i*width+j)+k]];
      }
    }
  }
```

The C code above is converted to Verilog in order to be synthesized in the hardware performing the transformation function.

```
SCAN:
```

```
begin
  if(read_valid == 1'b1)
  begin
    if(write_full == 1'b0)
    begin
      reg_r  <= reg_tran[wire_r[BIT_DEPTH-1:0]];
      reg_g <= reg_tran[wire_g[BIT_DEPTH-1:0]];
      reg_b <= reg_tran[wire_b[BIT_DEPTH-1:0]];
    end
    reg_valid <= 1'b1;
    state     <= SCAN;
  else begin
    if(read_done == 1'b1)
      state    <= IDLE;
     else
       state    <= SCAN;
    reg_valid <= 1'b0;
  end
end
```

4 Software Accelerator

An application code in C allows the FPGA configuration via the PCIe® driver [16].

This application opens a file that contains the image to be equalized, and then the driver opens communication with the on-chip memory in order to send the image. After that, the settings are sent to the slave module of the accelerator. These parameters contain the address location in the memory where the original image is stored and where the processed image will be saved, as well as the size of the pixel array. The next step is to initiate the accelerator computations when image extraction is completed, and finally, the image is extracted from the memory and stored in a file in the OS side.

The user application performs image equalization using the accelerator, as shown in Fig. 5.

Some registers must be configured in order to ensure the correct functionality and system parameters. Table 1 shows the correct PCIe® BAR0 configuration.

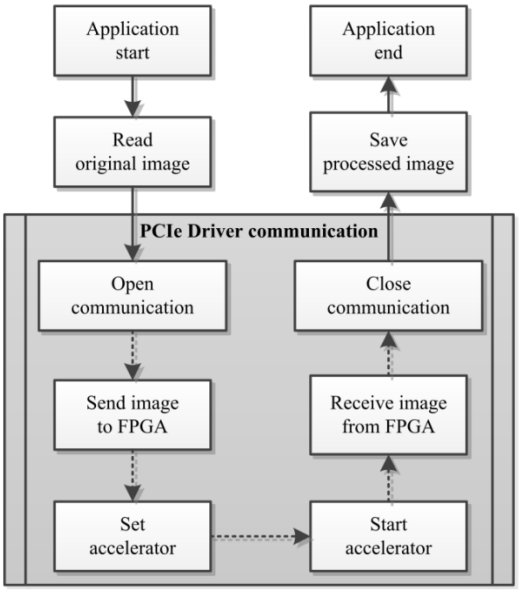

Fig. 5. Image processing flow diagram

4.1 PCIe® Channel Acquisition

These lines of code open the PCIe device driver node in the read and write mode; if the node cannot be opened, an error is printed in the terminal.

```
int main (int arg_count,char *arg_file_name[]) {
  int node;
  node = open("node_dir/altpciedev", O_RDWR);
  if (node == -1) {
    printf("Error When Open Node File\n");
    exit(1);
  }
```

4.2 Source Image Load to Memory

The following code opens the original image in the read mode and extracts the pointers to the image parameters and the pixel array, where the BITMAP element is a structure containing the image parameters from the BPM file:

```
// Open bmp file in read mode
  BITMAP b;
  FILE *fp; // Original image
  FILE *fo; // Preprocessed image
```

```
   fp = fopen(file, "rb");

// BMP Header
   fread(&b->typeb          1, 1, fp); // 0h
   fread(&b->typem,         1, 1, fp); // 1h
   fread(&b->size           4, 1, fp); // 2h
   fread(&b->reserved,      4, 1, fp); // 6h
   fread(&b->offset,        4, 1, fp); // Ah
   fread(&b->headersize,    4, 1, fp); // Eh
   fread(&b->width,         4, 1, fp); // 12h
   fread(&b->height,        4, 1, fp); // 16h
   fread(&b->planes,        2, 1, fp); // 1Ah
   fread(&b->bppx,          2, 1, fp); // 1Ch
   fread(&b->compress,      4, 1, fp); // 1Eh
   fread(&b->sizeim,        4, 1, fp); // 22h
   fread(&b->xpxpm,         4, 1, fp); // 26h
   fread(&b->ypxpm,         4, 1, fp); // 2Ah
   fread(&b->colors,        4, 1, fp); // 2Eh
   fread(&b->colors_imp,    4, 1, fp);//32h-35h

// Allocate color field memory
   b->stuff = (byte*) malloc ((dword)(b->offset-54));

// Color field
   fread(b->stuff, 1, b->offset-54, fp);

// Allocate data field memory
   b->data = (byte *) malloc  ((dword)(b->sizeim));

// Load Pixel Array into Memory
   fread (b->data, 1, b->sizeim, fp) != b->sizeim);
```

4.3 Accelerator Configuration and Image Processing

The following code shows the configuration of the accelerator through PCIe. First, the address of the on-chip memory is set, and the pixel array of data is sent. Second, the program sets the base address and the required parameters are set to start the accelerator. Third, the image is preprocessed from the on-chip memory, which is extracted and stored in a new image file.

```
   // Global pointers
   char *fout = "out_hw.bmp";
   dword *acc_buff;

   // Set mem_lo address
   ioctl(NODE, IOCTL_ADDR, 0x00000000);

   // System memory to on-chip memory
   write(NODE, b->data, b->sizeim, 0);

   // Memory allocation
   acc_buff = (dword *) malloc ((dword)(16*4))

   // Accelerator Configuration
   acc_buff[0]  = 0x00000000;    // Address Initial
   acc_buff[1]  = 0x00000000;    // Address Final
   acc_buff[2]  = b->sizeim / 4; // Length
   acc_buff[3]  = b->height;     // Pixel Height
   acc_buff[4]  = b->width;      // Pixel Width
   acc_buff[5]  = 0x00000001;    // Operation
   acc_buff[6]  = 0x00000001;    // Start
   acc_buff[7]  = 0x00000000;    // Process Time
   acc_buff[8]  = 0x00000000;    // Status

   // Set acc address
   ioctl(NODE, IOCTL_ADDR, 0x00020000);

   // Start mem_transfer
   write(NODE, acc_buff, (dword)(16*4), 0);
```

4.4 Storing Target Image

The following code extracts the array of processed pixels from the on-chip memory and the equalized image is obtained.

```
   // Memory Read
   // Set mem_hi address
   ioctl(NODE,    IOCTL_ADDR,    0x00000000);
read(NODE, b->data, b->sizeim, 0);

// Building the new image
// Write output file
   fwrite("BM                2, 1,        fo);
   fwrite(&b->size           4, 1,        fo);
   fwrite(&b->reserved,      4, 1,        fo);
   fwrite(&b->offset,        4, 1,        fo);
   fwrite(&b->headersize,    4, 1,        fo);
   fwrite(&b->width,         4, 1,        fo);
   fwrite(&b->height,        4, 1,        fo);
   fwrite(&b->planes,        2, 1,        fo);
   fwrite(&b->bppx,          2, 1,        fo);
   fwrite(&b->compress,      4, 1,        fo);
   fwrite(&b->sizeim,        4, 1,        fo);
```

```
  fwrite(&b->xpxpm,          4, 1,          fo);
  fwrite(&b->ypxpm,          4, 1,          fo);
  fwrite(&b->colors,         4, 1,          fo);
  fwrite(&b->colors_imp,     4, 1,          fo);
  fwrite(b->stuff,      1, b->offset-54,    fo);
  fwrite(b->data,       1, b->sizeim,       fo);

// Close files and release memory
  free(b->stuff);
  free(b->data);
  fclose(fp);
  fclose(fo);
```

5 Results

The transfer rate was evaluated and the driver sent the data blocks to read and write transactions achieving up to 5 MB/s read. Processing tests for the equalization algorithm were performed on the dual-core Atom™, and the accelerator design was tested in the FPGA reaching a frequency of 125 MHz.

The image tests consisted of using different frequencies and Atom™ configurations for processing two image sizes: with 400 x 300 pixels and 320 x 240 pixels. The accelerator has been tested with the same images resulting in less execution time than with the Atom™. Fig. 6 shows the resulting bar graph of times of the tests performed with two different image sizes and 8 Atom™ processor configurations.

In Table 2, the average latency of each configuration is reported for different image sizes.

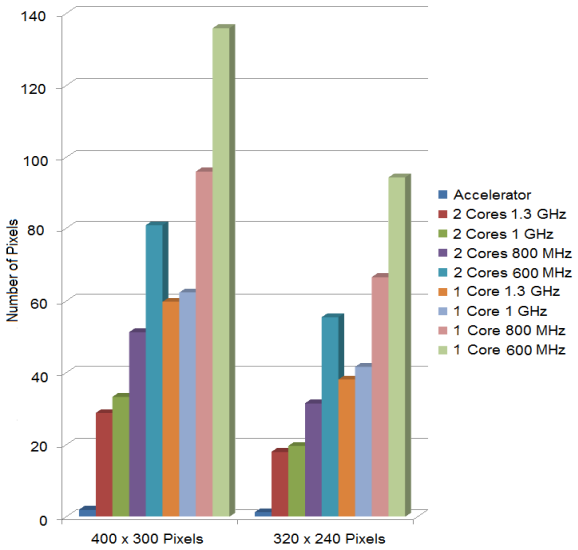

Fig. 6. Performance bar graph of algorithm executed with hardware acceleration and software with two different configurations and speeds

It is important to mention that the acceleration time is always accurate and depends directly on the size of the image plus an initial fixed time for configuration of the accelerator.

The image used for processing color equalization is presented in Fig. 7(a). This presents color saturation tones, so the image shows an overexposure. The driver sends the image to the accelerator, obtaining the histogram and the cumulative distribution function, which were used to obtain the transformation function in Fig. 7(b). Applying the transformation to each color tone in

Table 2. Average latency of image processing

Configuration	400 x 300 Pixels	320 x 240 Pixels	160 x 240 Pixels	80 x 60 Pixels
2 Cores 1.3 GHz	28.8330 ms	18.0239 ms	4.5659 ms	1.2589 ms
2 Cores 1 GHz	33.3200 ms	19.6569 ms	4.6270 ms	1.2780 ms
2 Cores 800 GHz	51.3199 ms	31.5179 ms	6.8320 ms	1.7770 ms
2 Cores 600 GHz	81.0969 ms	55.4729 ms	11.6070 ms	2.6820 ms
1 Core 1.3 GHz	59.7820 ms	38.1420 ms	4.5770 ms	1.2260 ms
1 Core 1 GHz	62.3409 ms	41.6739 ms	4.5799 ms	1.2509 ms
1 Core 800 MHz	96.0389 ms	66.6549 ms	14.2889 ms	1.7890 ms
1 Core 600 MHz	135.9160 ms	94.3860 ms	34.1359 ms	2.6920 ms
Accelerator	1.9256 ms	1.2344 ms	0.3127 ms	0.0824 ms

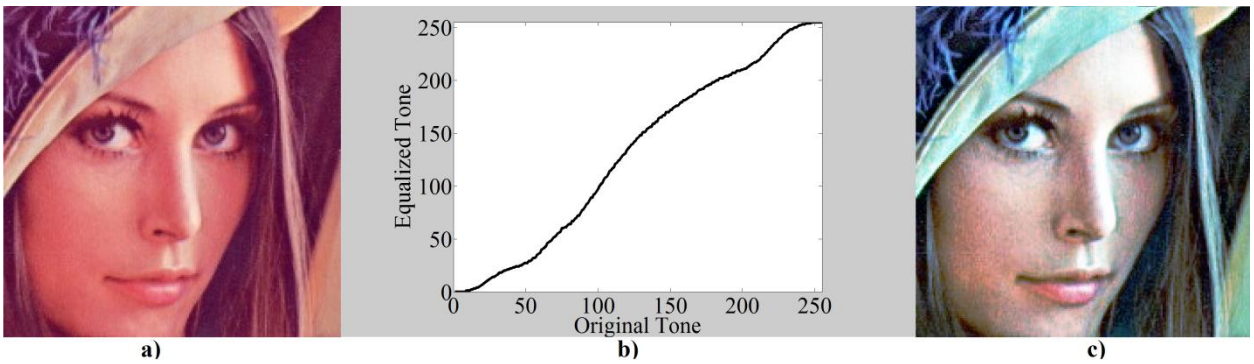

Fig. 7. Process for the equalization of a color image: (a) original image, (b) transformation function, (c) equalized image

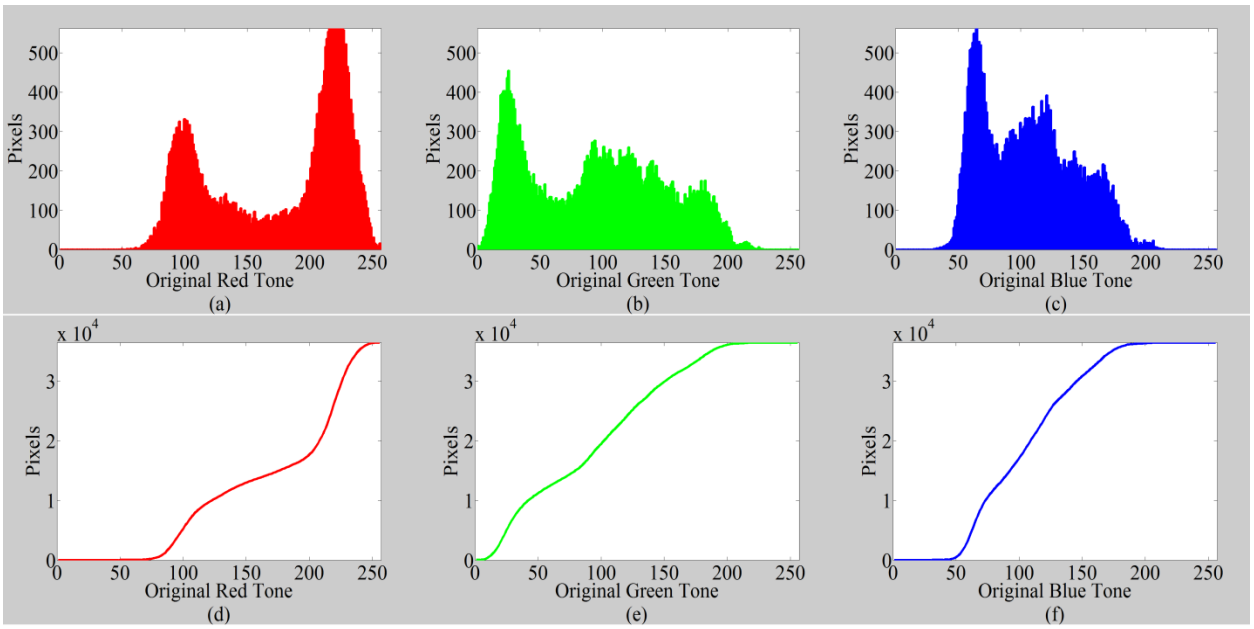

Fig. 8. Histograms and cumulative distribution functions of the original image: (a) histogram of red tone, (b) histogram of green tone, (c) histogram of blue tone, (d) cumulative distribution function of red tone, (e) cumulative distribution function of green tone, (f) cumulative distribution function of blue tone

the pixels of the image, the equalized image with even more distribution of colors is obtained as shown in Fig. 7(c).

Fig. 8 presents the histograms and the cumulative distribution functions for the tones of the original image. The histograms in Fig. 8(a), Fig. 8(b), and Fig 8(c) for red, green, and blue tones, respectively, show that color values of the pixels are distributed mainly in the area of highlights, and the image is considered as overexposed when it was captured.

The accumulation functions for each color—red, green, and blue—are shown in Fig. 8(d), Fig. 8(e), and Fig. 8(f), respectively.

The cumulative distribution functions are needed to compute the transformation function. In order to obtain image equalization, it is necessary to replace each pixel value with the new value from

Table 3. FPGA logic utilization

FPGA Components	Percentage Utilization	Component Utilization
Logic utilization	90 %	Routing
Combinational ALUTs	61 %	30,775/50,600
Dedicated logic registers	60 %	30,390/50,600
Pins	5 %	20/404
Block memory bits	81 %	3,716,208/4,561,920
DSP block 18 bits elements	1 %	4/312
GXB Receiver Channel PCS, PMA	50 %	4/8, 4/8
GXB Transmitter Channel PCS, PMA	50 %	4/8, 4/8
Total PLLs	25 %	1/4

the transformation function, obtaining a new image. In order to demonstrate that the new image has better equalization, the histogram and the cumulative distribution function obtained are shown in Fig. 9. The histograms for new red, green and blue tones are shown in Fig. 9(a), Fig. 9(b), and Fig. 9(c), respectively. It is easy to see that histograms are more uniform than those in Fig. 8(a), Fig. 8(b), and Fig. 8(c) for each color. The cumulative distribution functions shown in Fig. 9(d), Fig. 9(e), and Fig. 9(f) are more linear than the original functions from Fig. 8(d), Fig. 8(e), and Fig. 8(f), respectively.

The accelerator processing time for a color image of $M \times N$ pixels of 24 bit depth is $2 \times M \times N + 552$ clock cycles at 125 MHz after the accelerator has been initialized.

After implementation in an Arria® II GX FPGA with device part number EP2AGXE6XXFPGA, the logic utilization was 90% [17], where 30775 ALUTs and 4 DSP blocks were used for processing steps. 30,390 registers were used to allocate temporary data, and 3,716,208 memory bits were necessary in order to store all image data, as shown in Table 3.

It is mandatory that all routes between each data path be less than 8ns because the accelerator clock runs at 125 MHz in order to achieve the required data path time [18]. The processing elements were distributed in a pipeline with the following configuration: 256 steps to obtain the cumulative distribution function, 256 steps to obtain the transformation function, and 40 pipeline steps for integer division.

6 Conclusion

Preprocessed images offer a better distribution of colors, permitting the execution of facial recognition algorithms. This work includes the design, implementation, and verification of the hardware accelerator framework for image processing using the HW/SW platform, taking advantage of having a general purpose processor and a reprogrammable unit on the same integrated circuit, and facilitating the development of hardware accelerators. The PCIe® Protocol provides an optimum and scalable communication channel for transferring data between the processor and FPGA. The driver enables the control functions and management of sending and receiving data between the accelerator and the operating system, reaching a maximum transfer rate of 5MB per second. The hardware accelerator performs image processing in an interval less than a processor would require to obtain the same result via software. The reusable framework can develop different families of accelerators without the need to create a different communication protocol between the hardware and the software structure every time, thus allowing the user application and the processing algorithm to be modified. Using this

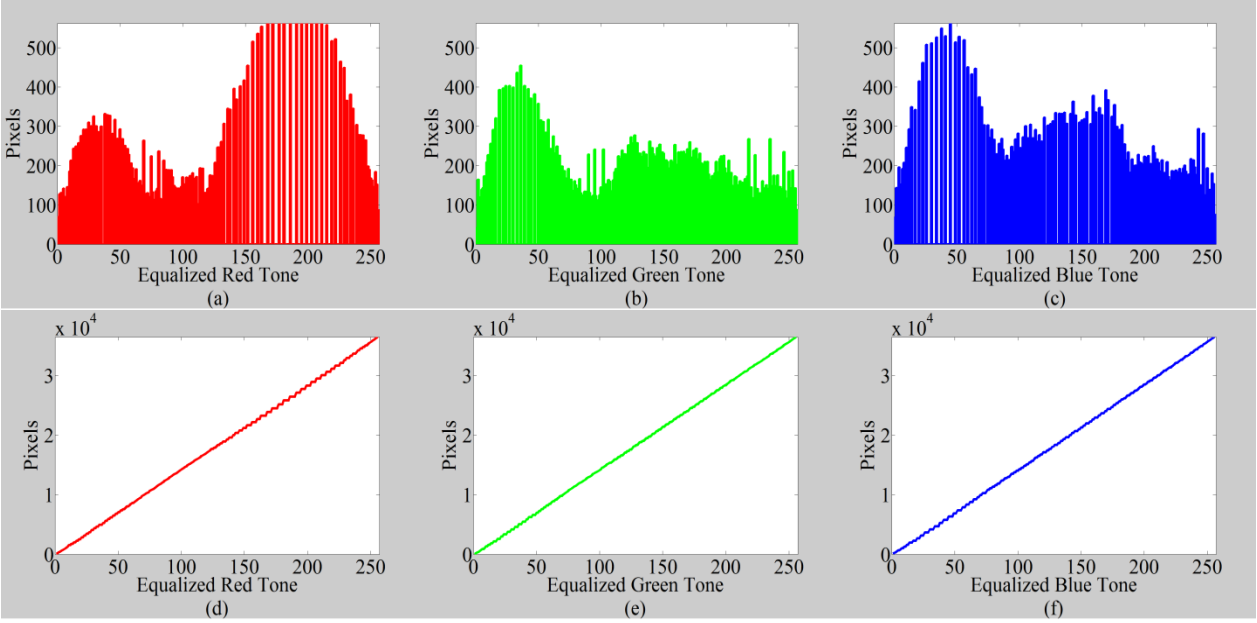

Fig. 9. Histograms and cumulative distribution functions of the equalized image: (a) histogram of red tone, (b) histogram of green tone, (c) histogram of blue tone, (d) cumulative distribution function of red tone, (e) cumulative distribution function of green tone, (f) cumulative distribution function of blue tone

framework saves time in the development of hardware accelerators.

Acknowledgment

The authors are grateful for the financial support provided in part by CONACYT (National Council of Science and Technology) as a doctoral fellowship grant.

References

1. **López-Juárez, I., Rios-Cabrera, R., Peña-Cabrera, M., Méndez, G.M., & Osorio, R. (2012).** Fast Object Recognition for Grasping Tasks using Industrial Robots. *Computación y Sistemas,* Vol. 16, No. 4, pp. 421–432.
2. **Ghassabeh, Y.A. & Moghaddam, H.A. (2007).** A Face Recognition System using Neural Networks with Incremental Learning Ability. *International Symposium on Computational Intelligence in Robotics and Automation (CIRA),* pp. 291–296, doi: 10.1109/CIRA.2007.382904.
3. **Marcus, G., Gao, W., Kugel, A., & Manner, R. (2011).** The MPRACE framework: An open source stack for communication with custom FPGA-based accelerators. *Southern Conference on Programmable Logic*, pp. 155–160, doi: 10.1109/SPL.2011.5782641.
4. **Cabrera, O., Oriol, M., Franch, X., Marco, J., López, L., Díaz, O. G. F., & Santaolaya, R. (2014).** Open framework for web service selection using multimodal and configurable techniques. *Computación y Sistemas,* Vol. 18, No. 4, pp. 665–682, doi: 10.13053/CyS-18-4-2057.
5. **PCI SIG. (2005).** *PCIe® Specification*. URL: http://www.pcisig.com/specifications/pciexpress.
6. **Wang, W., Bolic, M., & Parri, J. (2013).** pvFPGA: Accessing an FPGA based hardware accelerator in a paravirtualized environment. *International Conference on, Hardware/Software Codesign and System Synthesis (CODES+ISSS),* pp. 1–9, doi: 10.1109/CODES-ISSS.2013.6658997.
7. **Altera Corp. (2010).** *Avalon® Interface Specification*. URL: http://www.altera.com/literature/manual/mnl_avalon_spec_1_3.pdf.
8. **Intel (2010).** *Intel® Atom™ Processor E6x5C Series-Based Platform for Embedded Computing*.

9. **Altera Corp. (2012).** *Arria(R) II Device Handbook*. URL: http://www.altera.com/literature/lit-arria-ii-gx.jsp.
10. **Mencer, O., (2006).** ASC: a stream compiler for computing with FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 25, No. 9, pp. 1603–1617, doi: 10.1109/TCAD.2005.857377.
11. **Moreno, F., López, I., & Sanz, R., (2010).** A Design Process for Hardware/Software System Co-design and its Application to Designing a Reconfigurable FPGA. *13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD),* pp. 556–562, doi: 10.1109/DSD.2010.43.
12. **Corbet, J., Rubini, A., & Kroah-Hartman, G. (2005).** *Linux Device Drivers.* O'Reilly Media.
13. **Celik, T. & Tjahjadi, T. (2012).** Automatic Image Equalization and Contrast Enhancement Using Gaussian Mixture Modeling. *IEEE Transactions on Image Processing,* Vol. 21, No. 1, pp.145–156, doi: 10.1109/TIP.2011.2162419.
14. **Lucchese, L. & Mitra, S.K., (2001).** A new method for color image equalization. *International Conference on Image Proc.,* Vol. 1, pp. 133–136, doi: 10.1109/ICIP.2001.958971.
15. **Chauhan, R. & Bhadoria, S.S., (2011).** An Improved Image Contrast Enhancement Based on Histogram Equalization and Brightness Preserving Weight Clustering Histogram Equalization. *International Conference on Communication Systems and Network Technologies (CSNT),* pp. 597–600, doi: 10.1109/CSNT.2011.128.
16. **Young-Su, K. & Chong-Min, K. (2005).** Performance-driven event-based synchronization for multi-FPGA simulation accelerator with event time-multiplexing bus. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 24, No. 9, pp. 1444–1456, doi: 10.1109/TCAD.2005.852035.
17. **Ling, A.C., Singh, D.P., & Brown, S.D. (2007).** FPGA PLB Architecture Evaluation and Area Optimization Techniques Using Boolean Satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 26, No. 7, pp. 1196–1210, doi: 10.1109/TCAD.2007.891362.
18. **Gort, M. & Anderson, J.H. (2012).** Accelerating FPGA Routing through Parallelization and Engineering Enhancements. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (*Special Section on PAR-CAD 2010*),* Vol. 31, No. 1, pp. 61–74, doi: 10.1109/TCAD.2011.2165715.

Adrian Pedroza de la Crúz received the B.Sc. degree in Communications and Electronics Engineering from the University of Guadalajara, Jalisco, Mexico, in 2008, and the M.Sc. degree in Electronics and Computer Science Engineering from the University of Guadalajara in 2010. From 2010 to 2012 he worked at Intel® as a Component Design Engineer. Currently he is a Ph.D. student in Electrical Engineering at the Centre for Research and Advanced Studies of IPN, Zapopan, Jalisco, Mexico. Nowadays his research focuses in the area of parallel memory architectures for accelerators on hardware.

Miguel Ángel Carrazco Díaz currently is a Ph.D. Student in Electric Engineering at the Centre for Research and Advanced Studies of IPN, Zapopan, Mexico. He graduated from the University of Guadalajara, Mexico, in 2010 receiving the M.Sc. in Electronics and Computer Science Engineering. In 2008 he received the B.Sc. in Communications and Electronics Engineering. From 2010 to 2012 he worked as a Component Design Engineer at Intel® Labs Group in the Guadalajara Design Center. His current research work is focused on design of computer architectures for parallel processing and hardware acceleration.

Susana Ortega Cisneros received the B.Sc. degree in Communications and Electronics from the University of Guadalajara, Mexico, in 1990, the Master degree from the Center of Research and Advanced Studies of the IPN, Zacatenco, Mexico. Susana Ortega received her Ph.D. degree in Computer Science and Telecommunications from Autonomous University of Madrid, Spain. She specializes in design of digital architecture based on FPGAs, DSPs, and Microprocessors. The main lines of investigation in which she works are digital control, self–timed synchronization, electronic systems applied to biomedicine, embedded microprocessor design, digital electronics, and custom DSPs in FPGAs.

Juan José Raygoza Panduro received the B.Sc. degree in Communications and Electronics from the University of Guadalajara, Mexico, in 1989, the Master degree from the Center of Research and

Advanced Studies of the IPN, Zacatenco, Mexico. Juan José Raygoza received his Ph.D. degree in Computer Science and Telecommunications from the Autonomous University of Madrid, Spain. From 1996 to 2000, he worked in IBM, participating in the technological transfer of manufacturing hard disk heads at the IBM manufacturing plant San Jose C.A., Guadalajara, Mexico. He specializes in the design of digital architecture based on FPGAs, microprocessors, embedded system, and bioelectronics. The main lines of investigation which he works in are electronic systems applied to biomedicine, microprocessor design, digital control, embedded system.

Jorge Rivera Domínguez received the B.Sc. degree from the Technological Institute of the Sea, Mazatlán, Mexico, in 1999, and the M.Sc. and Ph.D. degrees in Electrical Engineering from the Centre for Research and Advanced Studies, National Polytechnic Institute, Guadalajara, Mexico, in 2001 and 2005, respectively. Since 2006, he has been with the University of Guadalajara, Guadalajara, Mexico, as a full-time Professor at the University Center for Exact Sciences and Engineering, Electronics Department. His research interests focus on regulator theory, sliding mode control, discrete time nonlinear control systems, and their applications to electrical machines. He has published more than 40 papers in international journals and conferences and has served as reviewer for different international journals and conferences.

Federico Sandoval Ibarra received the B.Sc. degree in Physics and Electronics from the UASLP in 1988, Mexico, and the Ph.D. in Electronics from INAOE, Mexico, in 1997. During 1991-1996 he was at the Microelectronics Laboratory of INAOE as a researcher developing wet-etching techniques and designing CMOS circuitry for silicon-based microsensors. In 1997 he was at CNM, Bellaterra (Spain), as a visiting researcher being involved in the development of surface micromachining techniques to design a fully integrated microphone. In 1999 he joined CINVESTAV, Guadalajara campus, Mexico. During 2002-2006 he was the coordinator of the Electronic Design Group. His research areas include A/D converters for multi-standard communications, design of VCOs for HF applications, and test of analog circuits.

Article received on 08/12/2014; accepted on 29/04/2015.
Corresponding author is Adrian Pedroza de la Crúz.