FPGA-Based Design Of a High-Performance and Modular Video Processing Platform

Christophe Desmouliers, Erdal Oruklu and Jafar Saniie
Department of Electrical and Computer Engineering
Illinois Institute of Technology
Chicago, Illinois 60616
Email: erdal@ece.iit.edu

Abstract— **In this paper, an FPGA-based design and implementation of a high-performance video processing platform (VPP) is presented. A hardware/software codesign system is proposed on Xilinx Virtex II Pro FPGA to realize complex algorithms for real-time image and video processing. This paper presents the framework of the VPP, discusses the architectural building blocks and FPGA synthesis results. Each hardware (custom accelerator blocks) and software (code running on an embedded CPU core) component is described comprehensively, laying the foundation for an adaptable and modular embedded system. As a case study, a real-time motion detection algorithm is implemented, demonstrating the feasibility of the proposed platform. Additional hardware accelerators can be easily plugged-in to the system for desired processing engines. VPP can be a robust, cost-effective solution for a broad range of multimedia applications including broadcasting and streaming video, video on-demand, video encoding/decoding, surveillance, detection and recognition.**

I. Introduction

Today, a very significant number of embedded systems focus on multimedia applications with almost insatiable demand on low-cost, high performance and low power hardware. However, image and video processing algorithms in intensive multimedia applications are computationally very complex, posing a serious challenge. Furthermore, a critical issue for many applications is *time-to-market* (TTM) which is the length of time it takes from a product being conceived until its being available for sale. TTM is especially important in industries where products are phased put quickly. Clearly, an inexpensive, high throughput, and adaptable architecture with rapid development cycle is desirable.

Therefore, in this study, an embedded HW/SW codesign platform based on a reconfigurable FPGA (Field Programmable Gate Array) architecture is proposed for video processing applications. The proposed platform uses conventional FPGA development tools, provides an adaptable, modular architecture for future-proof designs and shortens the development time of multiple applications with a single, common framework.

In the past, several platforms have been developed for multimedia processing such as DSP chips based on VLIW (very-large instruction word) architectures. DSPs usually run at higher clock frequencies compared to FPGAs, however, the hardware parallelism (i.e., number of accelerators, multipliers etc.) is inferior. More importantly, they are not as flexible and may not meet the demands of firmware updates or revisions of multimedia standards. The shortcomings of the DSP and general purpose processors led to more rapid adoption of reprogrammable hardware such as FPGAs in multimedia applications [1]. In [2], it has been shown that FPGA hardware can be used to achieve an 11x improvement compared to DSP hardware in runtime performance of H.263 video decoders. Authors in [3], proposed a prototyping framework for multiple hardware IP blocks on an FPGA. Their MPEG4 solution creates an abstraction of the FPGA platform by having a virtual socket layer that resides between the design and test elements which reside on desktop computers.A different approach [4], [5], uses a layered reconfigurable architecture based on partial and dynamical reconfigurable FPGA in order to meet the adaptivity and scalability needs in multimedia applications. In [6], instruction set extension is used for a motion estimation algorithm required in H.263 video encoder. Authors incorporate custom logic instructions into a softcore NiOS II CPU within an Altera FPGA. In our approach, designers can glue their custom logic to the existing framework by adding additional accelerator units (user peripherals). The development phase would be limited to these custom logic components.

In the following sections, we present the design of VPP, highlighting the flexibility provided to the user. A brief description of the video acquisition and conversion module by Xilinx is presented before describing the custom modules required for single frame and multi-frame video processing. Finally, we discuss a case study with a motion detection algorithm implementation using VPP.

II. System Overview

The primary board used for VPP is the Virtex-II Pro development board from DigilentInc [7]. This board has a Virtex-II Pro XC2VP30 FPGA with 30,816 logic cells, 136 18-bit multipliers, 2,448Kb of block ram (BRAMs) and two PowerPC processors. It has also a DDR SDRAM DIMM that can accept up to 2 GBytes of RAM. An expansion connector is available to equip the board with a VDEC1 video decoder that uses Analog Devices ADV7183B to sample the incoming analog video and convert it to digital values according to the video standards ITU-R.656 and ITU-R B.601. This board is ideal for a video processing platform since it has all the hardware

978-1-4244-3355-1/09/$25.00©2009 IEEE

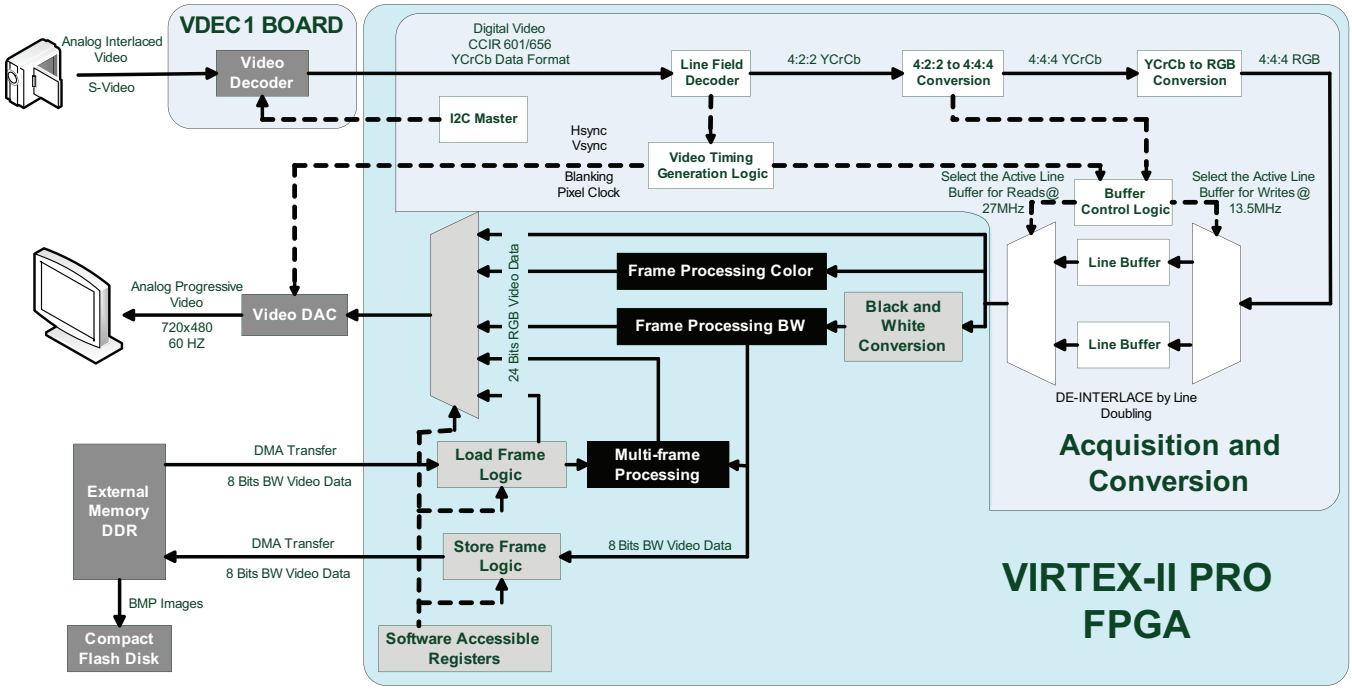

Fig. 1. Platform Design Overview

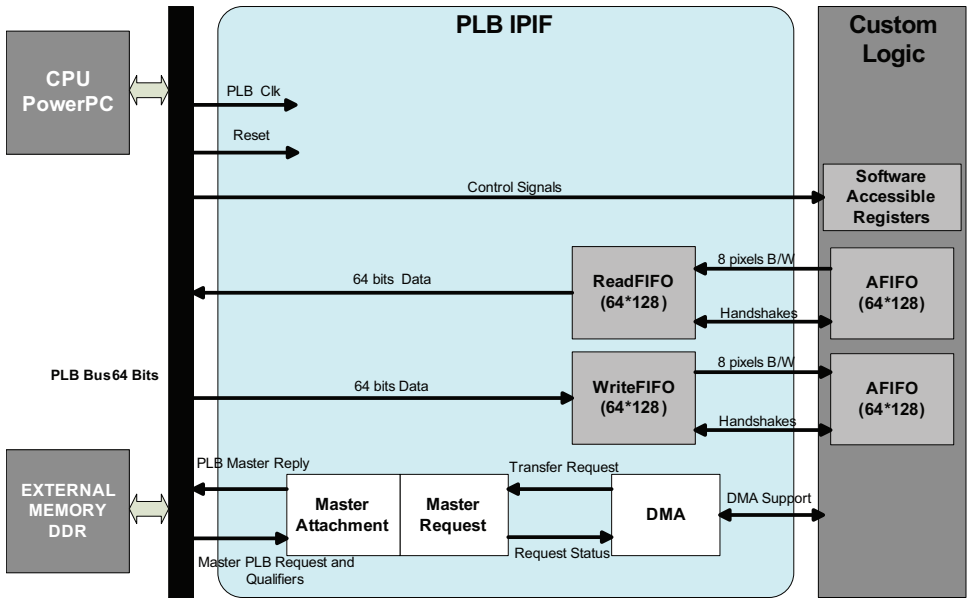

Fig. 2. Interface between custom logic and external memory

necessary to capture and display the data on a monitor using the XSGA video port. Moreover, a demonstration design is provided by Xilinx that displays the converted video data from the VDEC1 board on a standard VGA display as progressive video output. Using this design as a template, we build a flexible architecture that enables the user to perform real-time processing on a single frame or multiple frames. The overview of the proposed design is given Figure 1. It illustrates the multiple processing options available to the users:

- The user can choose to display the RGB video data without any processing.
- The user can perform real-time processing on a frame of RGB video data and display the output.
- The user can perform real-time processing on a frame of black and white video data and display the output.
- The user can save and retrieve frames to perform multi-frame processing and display the output.

Smoothing, edge detection filters are examples of single frame processing. Motion detection, object tracking are examples of multi-frame processing. These operations can be implemented

in hardware using custom FPGA logic. Hardware implementation blocks (frame processing units) are shown in black in Figure 1.

In the next section, video acquisition and conversion block provided by Xilinx is discussed.

III. Video acquisition and conversion system

The video data acquired from the VDEC1 board (see Figure 1) are in YCrCb format and hence, compatible with the standards ITU-R.656 and ITU-R B.601. These data are acquired serially at 27MHz. The acquisition and conversion module is composed of submodules: a line field decoder, conversion modules, line buffers and video timing generation module. The VDEC1 board is configured to capture the data on the S-Video input using the I^2C bus.

All video fields and line timing are embedded in video stream by using non-video related data values [8]. The purpose of the line field decoder is to detect the video format (NTSC or PAL) and extract the timing information. The 4:2:2 data stream will appear serially as:

$$Cb_{01}, Y_0, Cr_{01}, Y_1, Cb_{23}, Y_2, Cr_{23}, Y_3$$

From Figure 1 it can be seen that, a video frame will be extracted from the video stream using the signals from the Line Field Decoder. Cb_{01} and Cr_{01} share the color information for the first two pixels and Y_0 is the brightness information for the first pixel and Y_1 for the second pixel. Thus, Y information is available for all the pixels, while Cb and Cr appear for every 2 pixels. In order to obtain the 4:4:4 data, the Cb and Cr data need to be duplicated for the 2 pixels they correspond to. Hence the output data stream appears at half the frequency of the input stream which is 13.5MHz

The YCrCb data are then converted into RGB values and finally de-interlaced by line doubling. Data are written to a line buffer at 13.5MHz while the other one is read at 27MHz. Hence each buffer is read twice. For each frame, even and odd fields are separated by a vertical blanking field (interlaced video). For example, the odd field will be displayed as a complete frame by filling the missing lines with data of the previous line. Thus the rate will be double (60 frames per second instead of 30). Finally the video data are then synchronized to sync signals generated by the video timing generation logic and required for VGA video output. The implementation results of the stand-alone video acquisition and conversion module are given in Table I. The acquisition and conversion module uses very few resources of the FPGA, hence most of the FPGA fabric is available for additional custom logic in image and video processing applications.

TABLE I
SYNTHESIS RESULTS OF SYSTEM PROVIDED

Resource Type	Used	Available
Slices	247	13696
Slices FF	355	27392
4 input LUTs	291	27392
BRAMs	6	136
MULT18X18	5	136

IV. Custom Video Processing Hardware

Most of the algorithms used for video surveillance such as motion detection, object tracking, etc. require multiple frames to be stored and compared. The available block RAMs (BRAMs) of the Virtex-II Pro FPGA are not large enough to store a complete frame of 720x480. Hence, video frames need to be stored in the external memory. Furthemore, data transfers must be fast enough so that no data are lost and real-time processing can be done. To solve this problem Direct Memory Access (DMA) transfers have been used.

In order to perform DMA transfers to or from the external memory, the IPIF module [9] provided by Xilinx is used (Figure 2) with additional custom logic. Software accessible registers are used to send control signals to the custom logic to enable the *store block* or the *load block*. ReadFIFO is used to store data to the external memory and WriteFIFO to load data from the external memory. Asynchronous FIFOs (AFIFO) generated using CORE Generator [10] are needed because the clock used in the custom logic is different than the clock of the Peripheral Local Bus (PLB). For the store block, data are written to the AFIFO at 27 MHz and transferred to the ReadFIFO at 100 MHz.

Since the PLB is 64 bits wide, 8 pixels can be written (black and white) in parallel. Each line has 720 pixels so the depth of the FIFOs must be at least 90 (in this case, 128 has been chosen). Finally DMA transfers will be done line by line so 480 DMA transfers will be performed for a single frame.

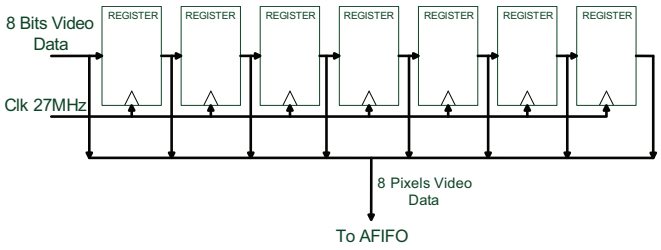

Fig. 3. Logic used to get 8 pixels video data for the AFIFO

A. Hardware required for storing frames

The incoming video data have to be written to the AFIFO when the user wants to store a frame. Registers are used so that 8 pixels can be written at a time (Figure 3). If the user wants to store a frame and the AFIFO is not full then the data are written to the AFIFO every 8 clock cycles. Finally if data are available in the AFIFO and the ReadFIFO is not full then those data are transferred to the ReadFIFO. Figure 4 and Figure 5 shows the state charts for these operations.

B. Software required for storing frames

A software component is required to setup and perform the DMA transfer. The flowchart of the C code executed by

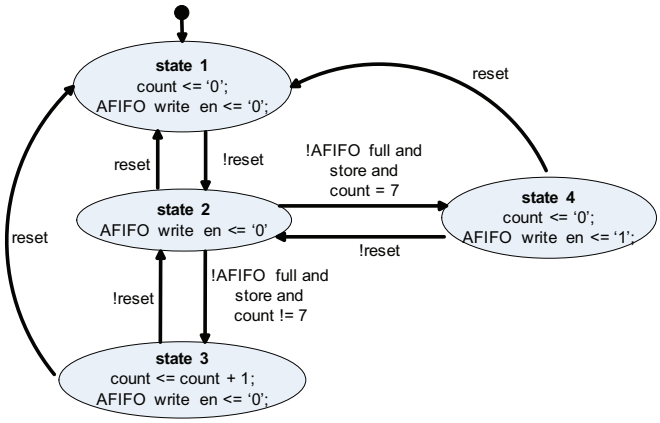

Fig. 4. State machine for writing data to AFIFO

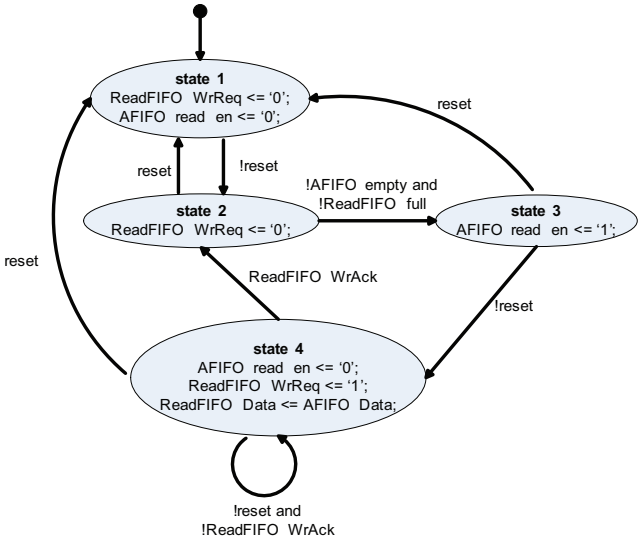

Fig. 5. State machine for transferring data from AFIFO to ReadFIFO

the PowerPC is given Figure 6. First, the user can select the number of frames to be stored in the external memory and the time between two frames. For example the user can specify to save 2 frames and skip 10 frames between those two. Then the AFIFO is reset in order to remove any data left from a previous frame transfer. The DMA control register are reset and initialized so that the source address is local (ReadFIFO) and the destination address is incremented (the external memory). Now the frames can be saved to the external memory by performing 480 DMA transfers. When the ReadFIFO contains enough data (720 bytes which correspond to 90 64bits wide data for one line) a DMA transfer is done, then the destination address is incremented and the next line can be transfered and so on. When all the frames have been transferred to the external memory, the process is completed and the function can be exited. The transfer is fast enough so that every single frame can be saved without any data loss (60 frames per second can be transferred). The same process can be used to load frames from the external memory to the logic and process them.

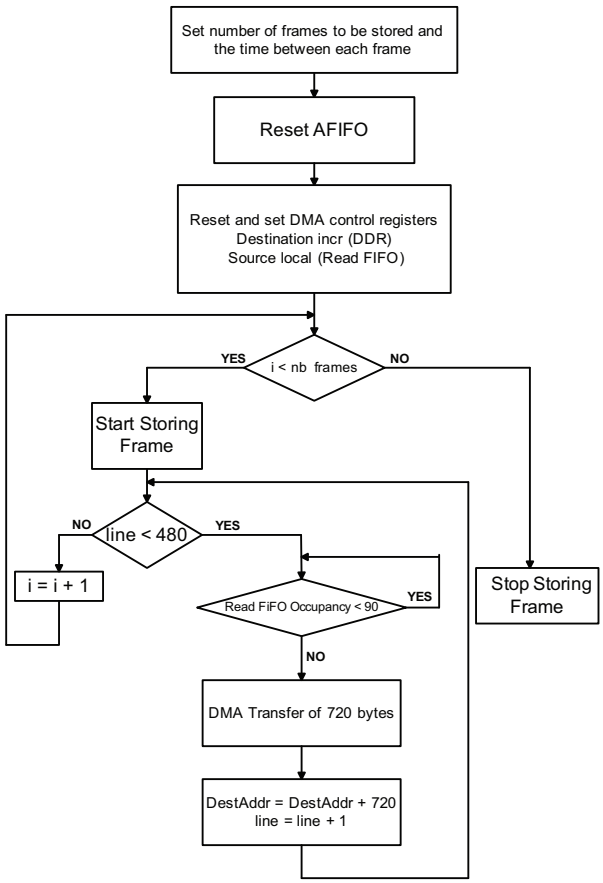

Fig. 6. Flowchart of the function for storing frames

C. Hardware required for loading frames

When data are available in the WriteFIFO, they need to be transferred to the AFIFO and hence to the state machine. Figure 7 shows the steps required for the transfer from the WriteFIFO to the AFIFO. If the AFIFO is not full and data are available in the WriteFIFO then a read request can be done. When the acknowledgment is received from the WriteFIFO, the data is ready to be written to the AFIFO. Finally the data are read every 8 clock cycles from the AFIFO if it is not empty (Figure 8). 8 pixels are read at a time so depending on the value of the counter each pixel can be retrieved.

D. Software required for loading frames

The C code for loading frame is similar to the one used to save frames. The flowchart of the C code is given in Figure 9. The user can specify the number of frames to be loaded. Then the AFIFO is reset. The DMA control registers are reset and initialized so that the source address is incremented (DDR) and the destination address is local (WriteFIFO). Finally 480 DMA transfers are completed (one for each line). If there is enough space available in the WriteFIFO to contain 720 bytes then the transfer can be done. The source address is incremented and the next line can be read and so on until the whole frame has been transferred.

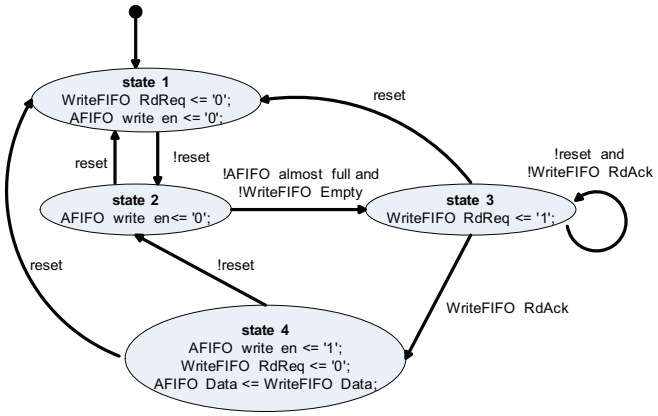

Fig. 7. State machine for transferring data from WriteFIFO to AFIFO

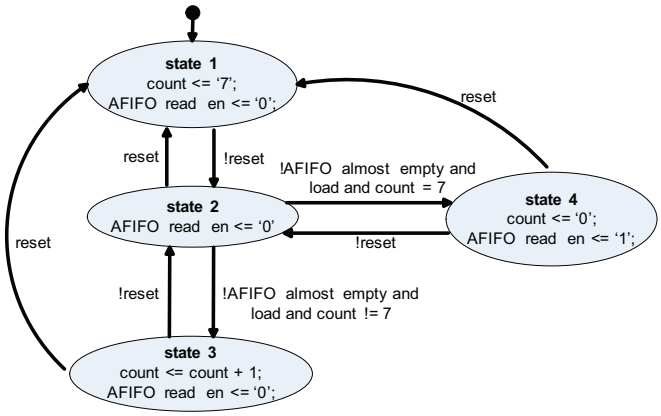

Fig. 8. State machine for reading data from AFIFO

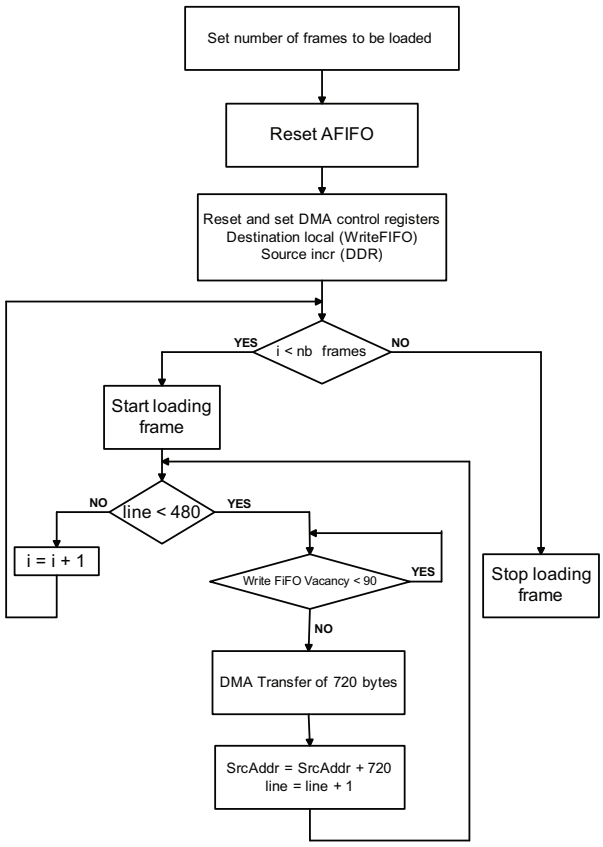

Fig. 9. Flowchart of the function for loading frames

E. Implementation results

The implementation results of the modified system are given Table II. It shows increased resource usage in terms of slices and LUTs due to the logic added and the IPIF module. Nevertheless only 10 BRAMs are used over 136, so that complex processing modules which require line buffers (BRAMs) can be added to the system.

V. A Case Study: Motion Detection Algorithm

As an application of the proposed video processing platform, a motion detection algorithm has been implemented. It is a typical example of multi-frame processing (see Figure 1) since the actual frame needs to be compared to a background image.

A. Hardware for the motion detection algorithm

For this algorithm (shown in Figure 10), we need a background image and the original image in which we want to detect the motion of the objects. First, the background image (a single frame) is stored to the external memory. This is done using the hardware and software presented in the previous sections. Then, this frame is compared to the current frame in black and white. A thresholding is applied to have only 0 or 255 values and reduce the sensibility to the noise. Finally, an edge detection algorithm (3 by 3 filter) is used and color values of the pixels of the actual frame are modified where an edge has been detected.

An example output of the algorithm is shown Figure 11. It can be seen that, motion (in this case a hand) has been successfully detected but also the object in the background (which is behind the hand) has disappeared. In order to remove this problem, the background needs to be updated regularly.

B. Software for the motion detection algorithm

The functions required for the algorithm are the similar to the ones presented in the previous sections. For the algorithm, the background is updated every 5 frames. Since the PLB (processor local bus) is used during the saving process of the background, no loading process can be done at the same time; hence actual data without processing are displayed during background update. The rest of the time, the background is loaded to the hardware and the motion detection algorithm is performed in real-time.

C. Implementation results

The implementation results of the system with a motion detection application are given Table III. Only 285 slices, 190 slices flip-flop and 499 4 input LUTs are added to VPP to perform a real-time motion detection algorithm. Hence, additional algorithms and processing engines can be implemented in parallel to VPP.

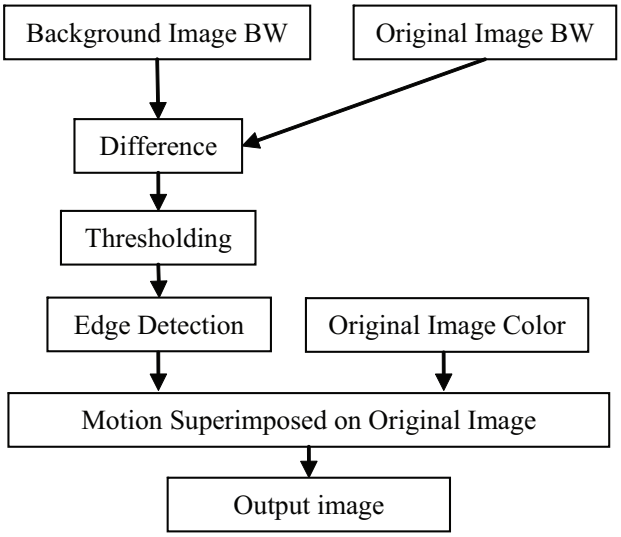

Fig. 10. Block diagram of the motion detection algorithm

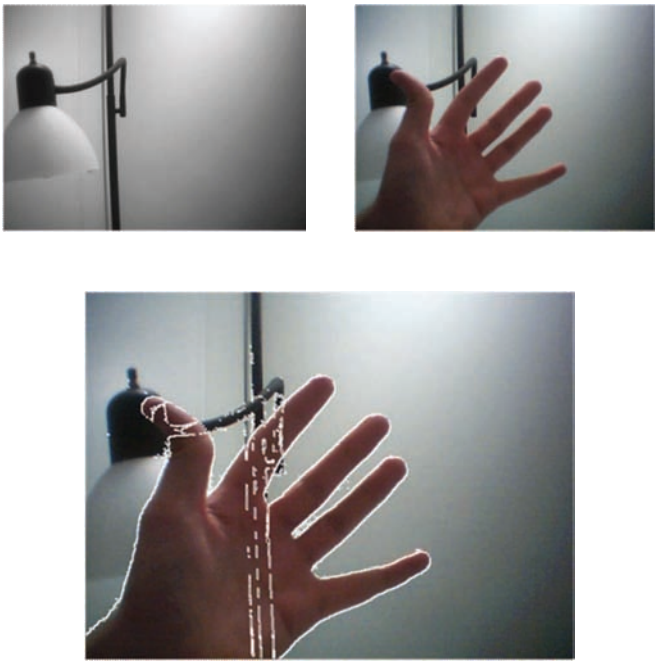

Fig. 11. Background image, Original image and Output image

VI. Conclusion

In this paper, we have presented a high-performance platform for real-time video processing on a Virtex-II Pro FPGA. This platform gives a lot of flexibility to the user for rapid development and testing of complex algorithms. As an example, JPEG or MPEG encoders can be implemented by hardware accelerators by using the proposed architecture to retrieve the frames to the custom logic. Moreover, dynamic partial self-reconfiguration can be used for very complex algorithms that require large resources. This would allow for temporal mapping of the algorithms as opposed to spatial mapping which may not be feasible for a single FPGA. The logic required can be reduced dramatically and multiple processing modules can be implemented on the same platform.

TABLE II

Synthesis results of custom system

Resource Type	Used	Available
Slices	1965	13696
Slices FF	2203	27392
4 input LUTs	3343	27392
BRAMs	10	136
MULT18X18	6	136

TABLE III

Synthesis results of motion detection application

Resource Type	Used	Available
Slices	2250	13696
Slices FF	2393	27392
4 input LUTs	3842	27392
BRAMs	10	136
MULT18X18	6	136

References

[1] L. Yi-Li, Y. Chung-Ping, and A. Su, "Versatile PC/FPGA-based verification/fast prototyping platform with multimedia applications," *IEEE Transactions on Instrumentation and Measurement*, vol. 56, no. 6, pp. 2425–2434, December 2007.

[2] M. Brogioli, P. Radosavljevic, and J. R. Cavallaro, "A general hardware/software co-design methodology for embedded signal processing and multimedia workloads," in *Fortieth Asilomar Conference on Signals, Systems and Computers, ACSSC '06*, Pacific Grove, California, October 2006, pp. 1486–1490.

[3] P. Schumacher, M. Mattavelli, A. Chirila-Rus, and R. Turney, "A software/hardware platform for rapid prototyping of video and multimedia designs," in *Proceedings of Fifth International Workshop on System-on-Chip for Real-Time Applications*, July 2005, pp. 30–34.

[4] X. Zhang, H. Rabah, and S. Weber, "Auto-adaptive reconfigurable architecture for scalable multimedia applications," in *Second NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2007*, Edinburgh, UK, August 2007, pp. 139–145.

[5] ——, "Cluster-based hybrid reconfigurable architecture for auto-adaptive soc," in *14th IEEE International Conference on Electronics, Circuits and Systems,ICECS 2007*, Marrakech, Morocco, December 2007, pp. 979–982.

[6] A. Atitallah, P. Kadionik, N. Masmoudi, and H. Levi, "HW/SW FPGA architecture for a flexible motion estimation," in *14th IEEE International Conference on Electronics, Circuits and Systems,ICECS 2007*, Marrakech, Morocco, December 2007, pp. 30–33.

[7] Xilinx, xupv2p reference designs. [Online]. Available: www.xilinx.com/univ/xupv2p_demo_ref_designs.htm

[8] DSP Design Line, ITU-R BT.656. [Online]. Available: www.dspdesignline.com/

[9] Xilinx, PLB IPIF Datasheet. [Online]. Available: www.xilinx.com/support/̃ip_documentation/plb_ipif.pdf

[10] Xilinx, CORE Generator. [Online]. Available: www.xilinx.com/products/design_tools/̃design_entry/coregenerator.htm