FPGA-Based Parallelism For Real-Time Video Image Processing

[1,2]Bethord A. B .Mahundi,[1]Xuewen Ding
Image Processing Theatre,School of Electronics Engineering
[1] Tianjin University of Technology and Education,China.
[2]BeM Technologies&Consultancy Center

Abstract

Image Processing is an important part of analyzing a scene in a video image and has application in many areas such as security monitoring and computer vision. It involves identifying places in an image where there is an abrupt change in intensity.Embedded Systems for Real-Time Video Image Processing must perform some operations faster than is possible since video data from a camera consists of a sequence of many frames, each of which is a rectangular array of many picture elements (pixels).Applications that involve less regularly structured data, or data that arrives at irregular intervals, are much harder to accelerate.We can accelerate performance of an operation by replication of hardware (Accelerator) resources to perform steps in parallel, up to the limits on parallelism implied by the data dependencies and the availability of data. It is possible to design custom FPGA-Based hardware (Accelerator) to provide parallelism and performing the operation at the required speed for real-time video image processing.

Keywords:Accelerator,Verilog,Field Programmable Gate Array,Video Image Processing,Graphical User Interface

1. Introduction

In recent years,much research has been performed that relies to various degrees on video image processing.A simple but accurate,effective and powerful accelerator that can be easily integrated and adapted for various applications is needed.Also,from technological standpoint,a flexible,has interface with camera,a link with Finite State Machine(FSM) and interface with GUI for controlling and monitoring the video image processing operation.Using Field Programmable Gate Array (FPGA) with a serial RS 232 interface to a PC with a graphical front-end.Such a flexible accelerator can be implemented and used in a variety of video image processing applications.In this paper we present the software and hardware details for FPGA-Based Parallelism for Real-Time Video Image Processing(Accelerator) which comprise:Image data and datapath computation,Image data pipeline, address generator,interfacing,Finite State Machine and Visual Basic Graphical User Interface.

2. System Architecture

Overview

In our development system of Parallelism for Real-Time Video Image Processing,we will adopt the approach of reading three rows of four adjacent pixels from the original image and storing them in registers,rather than including memories for whole rows.we will design the accelerator to process blocks of data, where a block consists of the three complete rows of the original image used to form a complete row of the derivative image. As we will see, processing a block involves a start-up phase, a repetitive sequence of computation, and a completion phase. These phases are repeated for each derivative image row.

The architecture for the accelerator datapath is shown in Figure 2. It is essentially a pipeline, with pixel data read from the original image entering into the registers at the top right, flowing through the 3 x 3 multiplier array on the left, then down through the adders to the Dx and Dy registers, then through the absolute value circuits and adder to the |D| register, and finally into the register at the bottom left. The

resulting derivative pixels are then written from that register to memory.(While a right-to-left data flow is opposite to usual practice, in this case, it has the advantage of preserving the same arrangement of pixels as that in an image.)

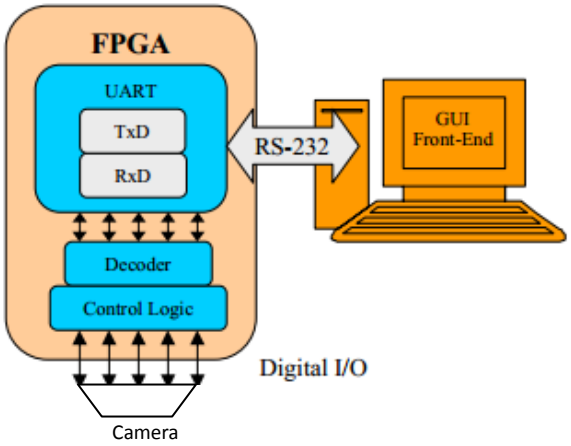

Figure 1:Parallelism for real-Time Image Processing System Architecture Diagram

The pipeline generates the derivative pixels for a given row in groups of four. The accelerator reads four pixels from each of the preceding, current, and next rows in memory into the three 32-bit registers at the top right of the figure1. Each register consist of four 8-bit pixel registers.Over the four subsequent clock cycles, pixels are shifted out to the left, one pixel at a time, into the multiplier array. Each cell in the array contains a pixel register and one or two circuits that multiply the stored pixel by a constant coefficient value. Since the coefficients are all -1, +1, -2, or +2, the circuits are not full-blown multipliers. Instead, multiplying by -1 is simply a negator, multiplying by +1 is a through connection with no circuitry, multiplying by -2 is a left shift of the result of a negator, and multiplying by +2 is simply a left shift. On each clock cycle, the array provides the partial products for a single derivative pixel,and the partial products are added and stored in the Dx and Dy registers.

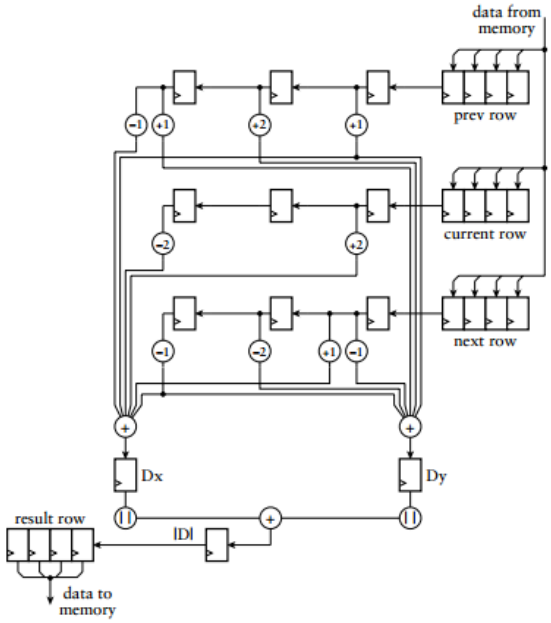

Figure 2:Architecture Diagram for Accelerator datapath

Also,on each clock cycle,the Dx and Dy values for the preceding pixel have their absolute values computed and added and stored in the |D| register. The resulting derivative pixel values are shifted into the result row register. When four result pixels are ready in the register, they are subsequently written to memory.

In the steady state, during processing of a row, the accelerator needs to write the pixels to memory from the result register before it can shift new pixels into the multiplier array and the Dx, Dy and |D| registers. Otherwise, the result values would be overwritten.Having written four pixels, the accelerator can push four more pixels through the pipeline,thus emptying the read registers and filling the result register. It can then write those result pixels and read in three more groups of four pixels, and repeat the process. However, this group of four pixel values is what we should write to the beginning of the derivative image row.The left-most position does not have a complete set of neighbors, so we don't compute a value for it.We will rely on the embedded software to clear that pixel value to 0 subsequently.

When we reach the end of a row, we need to drain the pipeline. Since the number of pixels in a row is a multiple of four (640 =160X 4),we can always read complete groups of four pixels each. After reading the last group, we perform four computation cycles normally. This gives us four result pixels to write, plus three remaining pixel values in the pipeline.We wish to finish the row by writing the four result pixels,

omitting the reads,performing four further computation cycles to drain the pipeline and shift the last pixel values into the required positions in the result register, and performing a final write. Note that this places an invalid value in the right-most result pixel register. This corresponds to the right-most pixel of a row, which does not have a complete set of neighbors. Again,we will rely on the embedded software to clear that pixel value to 0.

3. Implementation of Video Image Processing Technique

A.Verilog Modeling FPGA-Based Parallelism for Real-Time Video Image Processing

Verilog Modeling for Image Data computation

```
// Computational datapath
always @(posedge clk_i) // Previous row register
if (prev_row_load) prev_row <= dat_i;
else if (shift_en) prev_row[31:8] <= prev_row[23:0];
always @(posedge clk_i) // Current row register
if (curr_row_load) curr_row <= dat_i;
else if (shift_en) curr_row[31:8] <= curr_row[23:0];
always @(posedge clk_i) // Next row register
if (next_row_load) next_row <= dat_i;
else if (shift_en) next_row[31:8] <= next_row[23:0];
function [10:0] abs (input signed [10:0] x);
abs = x >= 0 ? x :-x;
endfunction
```

Verilog Modeling for Image data Pipeline

```
always @(posedge clk_i) // Computation pipeline
if (shift_en) begin
D = abs(Dx) + abs(Dy);
abs_D <= D[10:3];
Dx <= -$signed({3'b000, O[-1][-1]}) // ®C 1 * o[-1][-1]+ $signed({3'b000,O[-1][+1]}) // + 1 * o[-1][+1]- ($signed({3'b000, O[ 0][-1]}) // ®C 2 * o[ 0][-1] <<
1)+($signed({3'b000, O[ 0][+1]}) // + 2 * o[ 0][+1]<< 1)-$signed({3'b000, O[+1][-1]}) // ®C 1 * o[+1][-1]+$signed({3'b000, O[+1][+1]}); // + 1 * o[+1][+1]
Dy <= $signed({3'b000, O[-1][-1]}) // + 1 * O[-1][-1]+ ($signed({3'b000, O[-1][ 0]}) // + 2 * o[-1][ 0]<< 1)+ $signed({3'b000, O[-1][+1]}) // + 1 * o[-1][+1] - $signed({3'b000, O[+1][-1]}) // ®C 1 * o[+1][-1] - ($signed({3'b000, O[+1][ 0]}) // ®C 2 * o[+1][ 0] << 1)
 - $signed({3'b000, O[+1][+1]}); // ®C 1 * o[+1][+1]
O[+1][-1] <= O[+1][ 0];
O[+1][ 0] <= O[+1][+1];
O[+1][+1] <= next_row[31:24];
.....
end
always @(posedge clk_i) // Result row register
begin
if (shift_en) result_row <= {result_row[23:0], abs_D};
end
```

The first three always blocks in the module represent the three registers into which groups of four pixels are read from memory. Each block has a separate control signal governing loading, since the registers are loaded in successive memory read operations. They share a control signal for shifting, since they all shift a pixel out into the pipeline in parallel.

The next always block, represents the computational pipeline of the accelerator. The signals to which the block assigns, governed by the shift_en control signal, represent the pipeline registers. The signal O is a 3 x 3 array of pixel values, with indices corresponding to the difference in row and column numbers from those of the derivative pixel computed from the register values. For example, the element with indices [-1][+1] contains the pixel in the previous row and next column from the pixel being computed. Values are shifted into this array leftward from the left-most 8 bits of each of the input registers. The Dx and Dy values are computed from the array element values. In each case, the values are resized to 11 bits and converted to signed numbers, as we discussed earlier in our analysis of the precision requirements for the computation. Multiplying by 2 is performed with a logical shift left by one position, and multiplying by a negative coefficient is implemented by subtraction instead of addition. The absolute values of the Dx and Dy values, implemented by the abs function defined in the module, are added, and then scaled back from 11 to 8 bits to yield the final derivative pixel value.

The remaining always block represents the register that accumulates groups of four derivative pixels for writing to memory. Pixels are shifted into this register under control of the shift_en signal.

Verilog Modeling for Address Generator

There are several alternatives for deriving the read and write addresses, including maintaining counters for the image rows and columns. However, we can avoid the need to multiply by 640 by counting pixel offsets from the base addresses. In the case of the

original image, we start counting from an offset of 0 and increment by 1 for each group of four pixels read from memory. We add the offset to the base address to form the pixel-group address for the previous row. We add 640/4 to that to form the read address for the current row, and add 1280/4 to form the read address for the next row (assuming 00 for the least significant bits in both cases). In the case of the derivative image, we start counting from an offset of 640/4 and increment by 1 for each memory write.

```
always @(posedge clk_i) // 0 base address register
if (base_ce_0) base_0 <= dat_i[21:2];
always @(posedge clk_i) // 0 address offset counter
if (offset_reset) offset_0 <= 0;
else if (offset_cnt_en_0) offset_0 <= offset_0 + 1;
assign prev_addr_0 = base_0 + offset_0;
assign curr_addr_0 = prev_addr_0 + 640/4;
assign next_addr_0 = prev_addr_0 + 1280/4;
……..
always @(posedge clk_i) // D base address register
if (D_base_ce) D_base <= dat_i[21:2];
always @(posedge clk_i) // D address offset counter
begin
if (offset_reset) D_offset <= 0;
else if (D_offset_cnt_en) D_offset <= D_offset + 1;
end
assign D_addr = D_base + D_offset;
assign adr_o[21:2] = prev_row_load ? prev_addr_0 :
 curr_row_load ? curr_addr_0 :
 next_row_load ? next_addr_0 :
 D_addr;
assign adr_o[1:0] = 2'b00;
……..
end
```

The always blocks commented as being base address registers represent the base address registers for the original and derivative images, respectively. The always blocks commented as being address offset counters represent the counters for pixel groups read and written, respectively. The registers and counters are governed by control signals generated by the accelerator's control section. The adders are represented by the combinational assignments to the four address signals O_prev_addr, O_curr_addr, O_next_addr and D_addr. The assignment to the bus address signal adr_o represents the multiplexer that chooses among the generated addresses for memory read and write operations.

Verilog Modeling for Control Sequencing

We also need to sequence the accelerator's response as a bus slave when the user through GUI writes to the base address registers. Finally,we need to provide for synchronization with the embedded software controlling the accelerator. That requires some additional control and status registers, as follows:

A control register that, when written to, causes the accelerator to start processing an image. The value written is ignored.

A control register with an interrupt enable bit in bit 0.

A status register in which bit 0 is the done bit, set to 1 when the processor has completed processing an image. Other bits are read as 0. When the done bit is 1 and the interrupt enable bit is 1, the accelerator requests an interrupt. Reading the done bit has the side effect of acknowledging the interrupt and clearing the bit.

To keep the bus interface simple, we will map each of these registers at 32-bit aligned addresses.

REGISTER	OFFSET	READ/WRITE
Interrupt control	0	Write-only
Start	4	Write-only
Original image base address	8	Write-only
Derivative image base address	12	Write-only
Status	0	Read-only

Table 1:Register Map for the Accelerator

In each case, the accelerator can respond by setting ack_o to 1 in the next cycle, then back to 0 in the following cycle. We need to decode the bus address input to derive a select signal for the accelerator, and use the less significant address bits to determine which register to read or write.

In the case of a write to the start-register address, since there is no real register,we derive a control signal, start, that will be used by the accelerator control section to initiate a computation sequence.

```
assign start  = cyc_i && stb_i && we_i && adr_i ==
2'b01;
assign base_ce_0 = cyc_i && stb_i && we_i &&
adr_i == 2'b10;
assign D_base_ce = cyc_i && stb_i && we_i &&
adr_i == 2'b11;
…….
```

For write operations,we generate clock-enable signals using combinational logic.. For read operations,we form the data value to be returned to the GUI.The only real register is the status register, for which we return the value of the done bit,zero extended to 32 bits wide. For other register offsets, we just return all zeros.

The read value is multiplexed with the value of the result row register to drive the accelerator's data output bus, dat_o.

```
always @(posedge clk_i) // Interrupt enable register
if (rst_i)
```

```
int_en <= 1'b0;
else if (cyc_i && stb_i && we_i && adr_i == 2'b00)
int_en <= dat_i[0];
always @(posedge clk_i) // Status register
if (rst_i)
done <= 1'b0;
else if (done_set)
……
// This occurs when last write is acknowledged,
// and so cannot coincide with a read of the
// status register.
done <= 1'b1;
else if (cyc_i && stb_i && we_i && adr_i == 2'b00
&& ack_o)
done <= 1'b0;
assign int_req = int_en && done;
always @(posedge clk_i) // Generate ack output
ack_o <= cyc_i && stb_i && !ack_o;
……

// Data output multiplexer
always @(*)
begin
if (cyc_i && stb_i && !we_i)
if (adr_i == 2'b00)
dat_o = {31'b0, done}; // status register read
else
dat_o = 32'b0; // other registers read as 0
else
dat_o = result_row; // for master write
end
```

Finite State Machine Control an Accelerator

Verilog Modeling Finite State Machine

The control-section code includes declarations of internal signals for the control FSM, the row and column counters, and the control signals:

```
declarations of internal signals:
parameter [4:0] idle = 5'b00000,
                read_prev_0 = 5'b00001,
                ……..
reg [4:0] current_state, next_state;
reg [9:0] row; // range 0 to 477;
reg [7:0] col; // range 0 to 159;
wire start;
……
```

The row and column counters:

```
always @(posedge clk_i) // Row counter
if (row_reset) row <= 0;
else if (row_cnt_en)  row <= row + 1;
always @(posedge clk_i)  // Column counter
if (col_reset) col <= 0;
else if (col_cnt_en) col <= col + 1;
```

Blocks representing the finite-state machine:

```
always @(posedge clk_i) // State register
if (rst_i) current_state <= idle;
else current_state <= next_state;
```

A final always block combines both the state transition function and the output function into the one block.

```
always @* begin // FSM logic
offset_reset = 1'b0; row_reset = 1'b0;
col_reset = 1'b0;
row_cnt_en = 1'b0; col_cnt_en = 1'b0;
offset_cnt_en_0 = 1'b0; D_offset_cnt_en = 1'b0;
prev_row_load = 1'b0; curr_row_load = 1'b0;
……
endmodule
```

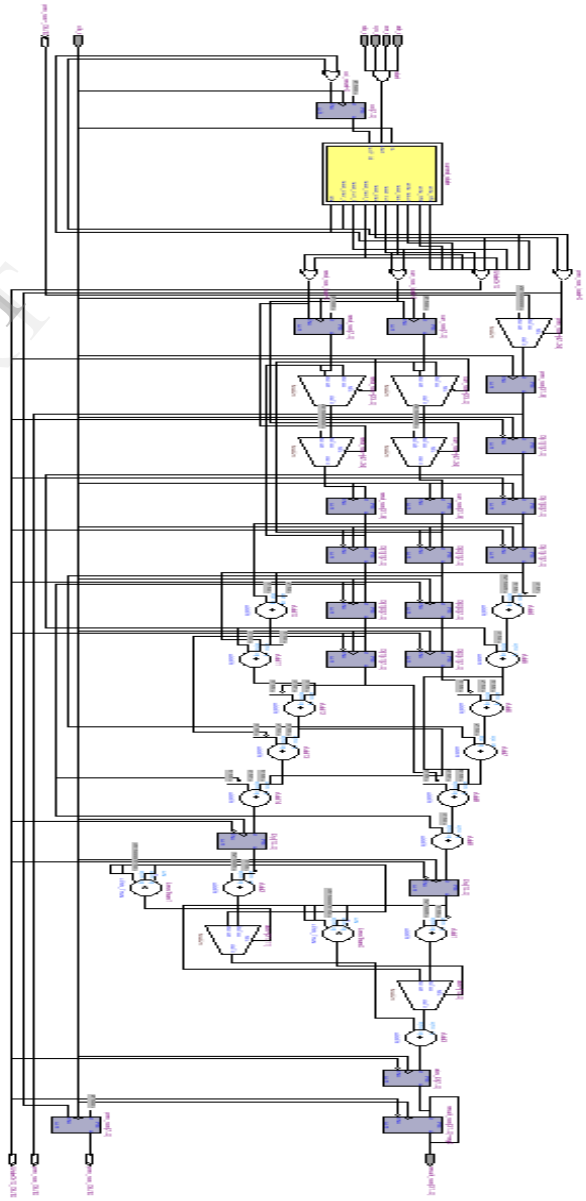

Figure 3: Register Transfer Level of the Implemented System

B. Camera Link Interface

The camera link standard has been devised to provide a generic 26 pin interface to a wide range of digital camera and as such we can specify a standard interface at the top level of my design. Although the interface requires 26 pins, they are configured differentially, and so we can specify the basic interface functionally using only 11 pins.There is a clock pin,which we can define as camera_clk,and then 4 camera control lines defined as cc1 to cc4, respectively. Using the 'camera_' prefix, we can therefore name these as camera_cc1,camera_cc2,camera_cc3 and camera_cc4. There are two serial communication lines serTFG (comms to frame grabber) and serTC (comms to camera) which we can name as camera_sertfg and camera_sertc, respectively. Finally, we have the 4 connection pins from the camera which will contain the data from the device and these are named camera_x0, camera_x1, camera_x2 and camera_x3.
Clearly, the actual interface requires differential outputs, and so eventually an extra interface will be required to translate the simple form of interface defined here to the specific pins of the connector.

C. Windows-Based Visual Basic GUI

A versatile GUI is designed using Visual Basic to give the user easy access to the FPGA and the attached hardware. Graphical interface design can be made especially easy with an integrated development environment (IDE), such as Microsoft's Visual Basic .NET, which was used to design the GUI. The code is entirely event-driven, and issues serial ASCII-encoded commands to change and monitor register values and variables on the FPGA according to user command (i.e. when a button is pressed or control value altered).

Figure 4:Graphical User Interface for FPGA control

The GUI provides many common input methods for changing variables. ON/OFF type functions can be controlled via push buttons which toggle FPGA registers on and off. The GUI also contains a recording and configuration loading function.The user can record a sequence of commands,objects image,time or variable changes and save this information to a configuration file. The file can be sent to the database through GPRS intended to watch transport motion, detect jams, and determine speed violation.Also the file can then be re-opened later which will load all of the exact same settings instantly onto the FPGA. This is useful when an FPGA is reprogrammed and the control code is altered. Using the load function, the user can easily set up the FPGA variables to the previous configuration or any previously stored preset.

D. Hardware Demonstration

The System Architecture described at previous section is using extensively to provide parallelism for high speed Real-Time image Processing applications.

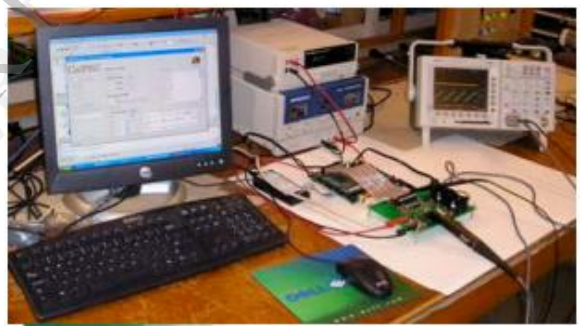

Figure 5:Hardware Implementation

Figure 5 shows the overall setup,with a Computer controlling ALTERA CYCLONE III EP3C26Q240C8N FPGA via RS-232.
Here we present details of application to a Road Traffic Monitoring for watching cars motion,detect jams,determine speed violation and recognize License Plate Registration.The output of the system is displayed on the visual basic GUI in figure 6 below(results section).

E. Results

Figure 6:Motion Tracking and Speed Measurement Display GUI

The figure 6 above shows the captured video image processed by the system by using the below explained verilog algorithm of analysed the scene intensity and coordinates and the results are displayed on the GUI.GUI displayed the speed of the car and its location.
Motion was filmed at the same road length for some car drives with known speeds $V_1<V_2<V_3<\ldots\ldots<V_n$.
Then every video sequence was processed to get passed distance S dependence from frame number calculated as asum of car displacements fixed in n previous sequence frames.

$S = a_1 + a_2 + a_3 + \ldots + a_n,$
Where a is a distance between car image geometrical centers in the I and i-1 frame:
$a_i = ((x_i - x_{i-1})^2 + (y_i - y_{i-1})^2)^{0.5}$

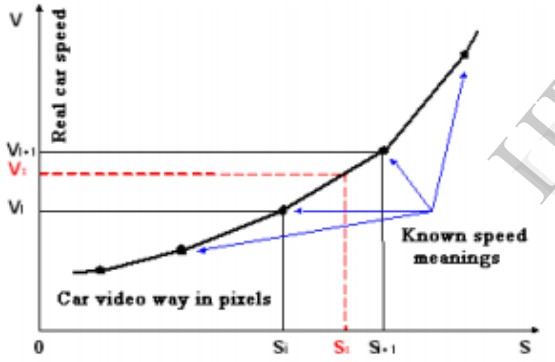

Figure 7:Function of Speed Approximation

Experimental curves $S_i = s_i(n)$ allow to estimate a speed of any car that moves within calibrated road segment.
Estimation is based on function v = v(s) for given frame number $n = n_o$
$V_x = A_i * S_x + B_i, \quad A_i = (V_{i+1} - V_i)/(S_{i+1} - S_i),$
$B_i = (V_i * S_{i+1} - V_{i+1} * S_i)/(S_{i+1} - S_i).$
Speed meaning may be corrected by v_x average for different n_0

4. Conclusion

In this paper,parallelism technique has been implemented,which provides performance of Real-Time video image processing operation steps in parallel,A parallel operation scheme that demonstrates a significant advantage over other conventional operation methods.This technique provides speed of processing of video image data faster than is possible.Its operation depend on the verilog algorithm you program to the FPGA chip.

References

[1] Douglas J Smith HDL Chip Design"A Practical guide for designing,synthesizing and simulating ASICs and FPGAs using VHDL or Verilog"Doone Publications,1996,pp.195-251.
[2] M.Schlett, "Trends in Embedded Microprocessor Design", Computer, vol. 31, no. 8, Aug, 1998,pp.44-49.
[3] Lipton, A.J. et al., "Moving target classification and tracking from real-time video",Applications of Computer Vision, 1998,pp.8-14. WACV '98. Proceedings., Fourth IEEE Workshop on, 1998.
[4] S. C. Chan, et al., "A programmable image processing
system using FPGA"
[5] Lou Tylee "Learn Visual Basic 6.0"1998,pp 7-10.
[6] www.altera.com

Bethord A.B.Mahundi
He received Bachelor's Degree of Science in Electronics and Communication Engineering fromSt.Joseph College of Engineering and Technology,Tanzania in 2008. He is currently pursuing a Master's Degree of Science in Electronics Engineering at Tianjin University of Technology and Education,China,Where his field of specialization is in Information Processing and Embedded Systems.

Xuewen Ding

He received Master's Degree of Science in Engineering and Ph.D.Degree in Signal and Information Processing from Tianjin University,China in 2005 and 2008 respectively.He has acted as a primary role in multi-projects such as National Nature Science Foundation of China,Key Nature Science Foundation of Tianjin and Tianjin Science and Technology Fund Planning Project.He has published over ten papers in video and image processing and applied for two patents.His current interest researches include digital image processing,video processing and transmission and machine vision.He has been teaching Information Theory and Code and Digital Image Processing to graduate students at Tianjin University of Technology and Education.