HARDWARE/SOFTWARE CO-DESIGN OF A 2D GRAPHICS SYSTEM ON FPGA

Kahraman Serdar Ay[1] and Atakan Doğan[2]

[1]TUBITAK BILGEM, Kocaeli, Turkey
serday.ay@bte.tubitak.gov.tr
[2]Dept of Electrical and Electronics Engineering, Anadolu University, Eskisehir, Turkey
atdogan@anadolu.edu.tr

ABSTRACT

Embedded systems in several applications require a graphics system to display some application-specific information. Yet, commercial graphic cards for the embedded systems either incur high costs, or they are inconvenient to use. Furthermore, they tend to quickly become obsolete due to the advances in display technology. On the other hand, FPGAs provide reconfigurable hardware resources that can be used to implement graphics system in which they can be reconfigured to meet the ever-evolving requirements of graphics systems. Motivated from this fact, this study considers the design and implementation of a 2D graphics system on FPGA. The graphics system proposed is composed of a CPU IP core, peripheral IP cores (Bresenham, BitBLT, DDR Memory Controller, and VGA) and PLB bus to which CPU and all peripheral IP cores are attached. Furthermore, some graphics drivers and APIs are developed to complete the whole graphics creation process.

KEYWORDS

Accelerator architectures, computer graphics, digital circuits, embedded software

1. INTRODUCTION

Both computing and graphics needs of embedded systems have been growing in many areas such as automotive, defense, and GSM. Nowadays, addressing these needs within a power budget is one of the major challenges for the embedded systems [1], [2]. As far as meeting the embedded systems' graphics needs is concerned, there are a few hardware solutions: For the applications with high-end graphics requirements, a single board computer can be joined with a separate graphics expansion board over PCI or PCI-e bus. Or, a hybrid architecture with a microprocessor and graphics-processing unit can be adopted [3]. For the lower end applications, a reconfigurable hybrid architecture that deploys a microprocessor for programmability and an FPGA for low power and high performance hardware graphics acceleration can be preferred [4]-[10].

In [4], the suitability of FPGAs for implementing the graphics algorithms was evaluated based on three different graphics algorithms. It was then found that FPGAs could reach a performance level between the custom graphics chips and general processors with specialized graphics instruction sets. Though, FPGAs have a key advantage of being flexible in that it can be reconfigured to implement various graphics algorithms as required.

In [5], a reference design for the automotive graphics systems was introduced, which supports some 2D graphics operations in hardware.

According to [6], the graphics chips become obsolete in less than two years, which makes supporting the military systems with integrated graphical displays over many years a major challenge. In order to protect the longevity of these systems, a 2D graphics engine on FPGA was proposed. The engine supports only Bresenham Line Generator and a few simple BitBLT (Bit Block Transfer) operations.

Similar to the military systems, in [7], FPGA based graphics systems were recommended for the automotive systems to keep up with the advances in display technology. Different from [6], the display modules instead of graphics chips were identified to be the part with short lifetime.

In [8], [9], and [10], some basic 3D graphics algorithms were implemented on FPGA. Common to these studies, they only focus on the design of graphics hardware. This study, on the other hand, considers not only the hardware design but also the software design of a 2D graphics system.

On the market, there exist relatively expensive IP cores which support 2D and 3D graphics algorithms, e.g. [11], [12], and [13]. The existence of these IP cores proves the necessity of external graphics peripherals and the potential importance of FPGA technology for the graphics systems.

The Khronos Group has provided OpenGL ES specification that is a low-level, lightweight API for advanced embedded graphics using well-defined subset profiles of OpenGL [14]. Several recent studies [15]-[18] have further focused on embedded graphics systems.

In this study, the hardware/software co-design principles are followed to design a 2D graphics system on FPGA from which many low-end applications are expected to benefit. The proposed system is composed of a CPU IP core, some peripherals that include Bresenham, BitBLT, DDR Memory Controller, and VGA IP cores, and PLB bus to which CPU and all peripheral IP cores are attached. Bresenham and BitBLT peripherals, which are designed and realized in this study, implement some computationally intensive 2D graphics operations in hardware. In the proposed system, the co-operation of IP cores, graphics drivers and APIs is exploited to support all graphics operations. The drivers and APIs that are implemented in C programming language will run on CPU IP core; they initialize the modeling and rendering stages of the graphics creation process; they manage the process by driving the related IP core peripherals.

The rest of the paper is organized as follows: Section 2 briefly introduces the fundamentals of some graphics algorithms realized in IP cores developed in this study. Section 3 and 4 give the details of the hardware and software design of the proposed graphics system. Section 5 provides some evidence to demonstrate the system operation. Finally, Section 6 concludes the study.

2. GRAPHICS SYSTEM ALGORITHMS

The graphics generation process in 2D consists of three main phases: geometric modeling, rendering, and monitoring [19]. In the geometric modeling phase, the synthetic models of objects are generated based on some geometric primitives (points, lines, circles, polygons, etc.). The first phase is realized by a set of APIs available in the system's software as explained in Section 4.

During rendering, the synthetic models due to the first phase are processed through 2D rendering pipeline in order to obtain the rendered images of real objects. The graphics systems' 2D rendering pipeline has also three phases. (i) Modeling transformation: This phase corresponds to transforming the geometric primitives into the real coordinate system based on translation, scaling, rotation and shearing transformation functions. (ii) Viewing transformation: It is the phase in which the real coordinates of the geometric primitives' corner points are transformed into the screen coordinate system. Both modeling and viewing transformations are handled by the system's software as explained in Section 4. (iii) Rasterization: The phase aims at finding the pixel representation of all geometric primitives that form an image. In this study, in order to perform 2D rasterization, Bresenham's Line Algorithm and Scan Line Polygon Filling Algorithm are implemented.

Rendered images are stored in a special memory area called framebuffer in a graphics system. Finally, the monitoring phase is required to deliver the rendered images inside framebuffer to a monitoring device (a VGA monitor).

2.1. Bresenham's Line Algorithm

Bresenham algorithm requires basic assembly level instructions such as addition, subtraction and bit shifting for its implementation. Thus, it is suitable for a high-speed implementation and relatively independent from the underlying hardware architectures [19]. Based on these promising features, in this study, Bresenham algorithm is chosen to draw lines, and it is implemented by the Bresenham IP core as explained in Section 3.

2.2. Polygon Filling

A polygon is composed of a finite sequence of straight-line segments (edges) and vertices where two edges meet. Polygon filling corresponds to painting two dimensional area enclosed by the polygon. In this study, Scan Line algorithm is employed for the polygon rasterization [19]: (i) Pre-process the polygon to be filled by shortening one of the edges that meet at any polygon vertex pixel by one pixel so that every vertex pixel in the polygon cut-pixel list will always be related to a single edge. (ii) Determine all horizontal lines that cut the polygon. (ii) For each horizontal line, find polygon cut-pixels. (iii) Paint those pixels that are between these cut-pixels and belong to the polygon.

Scan Line algorithm is implemented by means of API functions in Polygon API and BitBLT IP core that is driven by these APIs. BitBLT IP core is used for filling the horizontal line pieces inside the polygon. The implementations of BitBLT IP core and Polygon API are detailed in Section 3 and 4, respectively.

2.3. BitBLT Operations

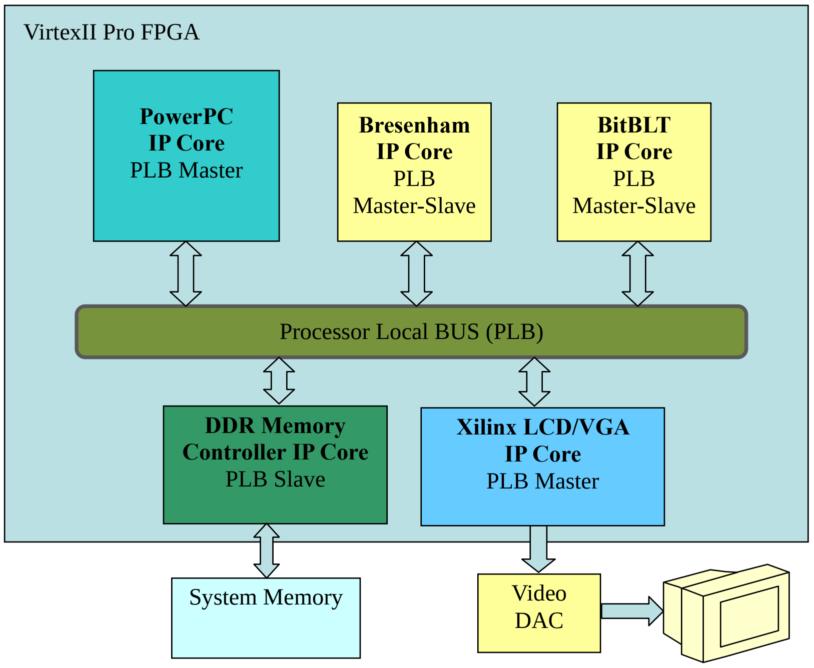
Before the rasterization, the vector graphics operations, such as modeling and coordinate transformations, only deal with the corner points of object models. After the rasterization, every object is represented by a bitmap (a set of pixel values) and after-rasterization operations on bitmaps become possible. The after-rasterization operations, which are the highest cost graphical ones during the graphics generation, include transporting bitmaps, applying Boolean operators, obtaining composite images, etc. [11]. In this study, BitBLT IP core is designed for the after-rasterization operations.

2.4. Alpha Composition

In the ARGB (Alpha, Red, Green, Blue) color model [20], the alpha composition operation is utilized to obtain the pixel values of a composite image. Specifically, it is applied to any pixel as follows [21]: For a given pixel, if there is only object A with color C_A and alpha value α_A, the color of this pixel becomes $\alpha_A C_A$. If another object B with color C_B and alpha value α_B coincides over this pixel with object A (B over A); the contribution of object B to this pixel is $\alpha_B C_B$ and object B allows only $(1-\alpha_B)$ percent of the pixel area to be transparent to any object behind it. As a result, the final color value of the pixel can be calculated as $(\alpha_B C_B+(1-\alpha_B)\alpha_A C_A)$. In this study, the alpha composition is implemented in BitBLT IP core.

3. GRAPHICS SYSTEM'S HARDWARE DESIGN

The graphics system design is split into two main tasks, namely hardware design and software design. During the hardware design, two IP cores are developed for several graphics algorithms. Furthermore, some mechanisms to provide communication between CPU and IP cores and to drive the peripherals with the help of the software running on CPU are developed. During the software design, the driver functions of graphics IP cores and several graphics APIs are implemented.


VirtexII Pro FPGA
PowerPC IP Core PLB Master
Bresenham IP Core PLB Master-Slave
BitBLT IP Core PLB Master-Slave
Processor Local BUS (PLB)
DDR Memory Controller IP Core PLB Slave
Xilinx LCD/VGA IP Core PLB Master
System Memory
Video DAC
Figure 1. The hardware architecture of the graphics system.

The hardware architecture of the proposed system is shown in Fig. 1. In order to implement this system, Xilinx EDK platform is used. Xilinx EDK makes it possible to use PowerPC CPU hard IP core featured in Virtex II FPGA, run application software on PowerPC core, exploit some ready-to-use free Xilinx IP cores (DDR Memory Controller and LCD/VGA), and attach custom logic blocks (Bresenham and BitBLT IP) and all other IP cores to Processor Local Bus (PLB bus) as peripherals. In the following sections, the system's hardware design is explained in detail.

3.1. Attaching IP Cores to PLB

Processor Local Bus, which is a part of CoreConnect bus architecture developed by IBM for SoC designs [22], is the bus architecture of choice in the proposed design. In order to attach a custom logic block as a peripheral to the bus, EDK provides PLB IP Interface (PLB IPIF) [23], which is an interface between PLB bus and a user IP core. Furthermore, Xilinx IPIC (IP Interconnects), as a part of PLB IPIF, includes the signal definitions that form the interface between PLB IPIF and a custom logic. In this study, in order to connect a custom IP core to PLB IPIF, the hardware architecture shown in Fig. 2 is used.

According to Fig. 2, connecting a custom logic to the bus require three basic interface components (slave_read_module, slave_write_module and master_module) and a set of memory mapped registers. In Fig. 2, the interface components only have an interface with IPIC. Thus, all communication between an IP core and the rest of the system must go through these modules.

Furthermore, the modules provide a set of memory-mapped registers with IP core: slave_read_module meets the read requests from PLB and makes all memory mapped registers reachable by other system components; slave_write_module satisfies the write requests from PLB and enables others to write into the memory mapped read-write registers.

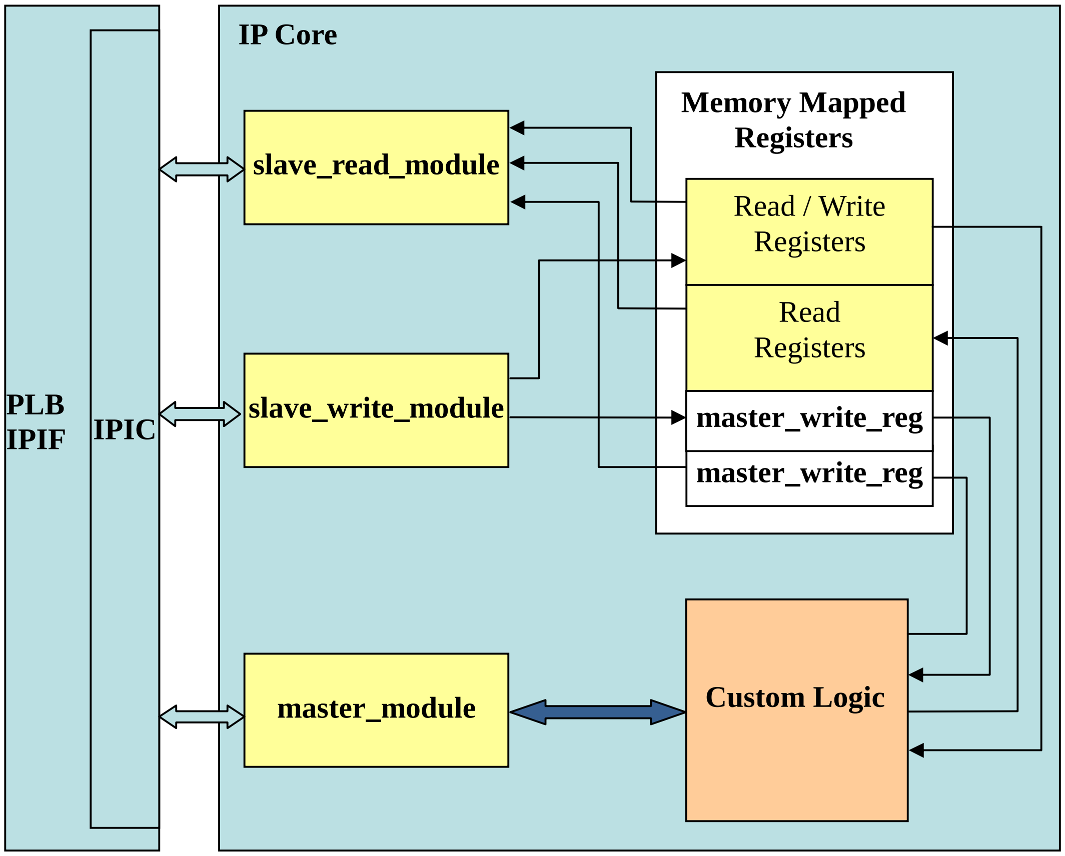


Figure 2. Attaching a custom logic IP core to PLB bus.

When a custom logic writes data to an external memory location (a memory address not in its address space), it interacts with PLB IPIF as follows: (i) The logic puts the data to be written into a special register (master_write_reg) while providing a write strobe and destination address with master_module. (ii) master_module initiates a write request to PLB IPIF where the source address (address of master_write_reg) and destination address are specified in IP2IP_Addr and Bus2IP_Addr IPIC signals, respectively. (iii) PLB IPIF makes a read request to slave_read_module with the address specified by IP2IP_Addr signal. (iv) PLB IPIF reads the content of master_write_reg. (v) PLB IPIF writes this data to the destination address, waits for an acknowledgement from the destination IP core. (vi) After receiving the acknowledgement, PLB IPIF sends an acknowledgement signal to master_process, which completes the write operation.
The scenario in which a custom logic asks master_module for reading from an external address is realized as follows: (i) The logic provides a read strobe and an external source address with master_module. (ii) master_module initiates a read request to PLB IPIF where the source address and destination address (address of master_read_reg) are specified in Bus2IP_Addr and IP2IP_Addr IPIC signals, respectively. (iii) PLB IPIF reads the content of source address. (iv) PLB IPIF provides this externally read data with slave_write_module, which writes it into master_read_reg. (v) Once the write operation is completed, slave_write_module sends an

acknowledgement to PLB IPIF. (vi) After receiving the acknowledgement, PLB IPIF sends an acknowledgement signal to master_process, which completes the read operation.

3.2. Bresenham IP Core

Bresenham IP core is attached to PLB bus by means of PLB interface design mentioned in Section 3.1. Bresenham IP core implements a modified version Bresenham algorithm in [19] in bresenham_module (custom logic in Fig. 2) and a set of memory-mapped registers specific to bresenham_module. The proposed new implementation of Bresenham algorithm with eight input parameters is shown in Fig. 3.

```
function bresenham (major_y, major_axis_first, major_axis_last,
            minor_axis_last, error_major_shifted, error_minor_shifted,
            negative_direction, rgb)
{
  error=0;

  if (negative_direction==TRUE) addDirection=1 else addDirection=-1;

  while (major_axis_first <= major_axis_last)
  {
    if (major_y==TRUE)
      fillPixel(rgb, minor_axis_first, major_axis_first);
    else
      fillPixel(rgb, major_axis_first, minor_axis_first);

    if (error>=0)
    {
      minor_axis_first= minor_axis_first + addDirection;
      error= error – error_major_shifted;
    }

    error= error + error_minor_shifted;
    major_axis_first= major_axis_first + 1;

  }
}
```

Figure 3. Bresenham algorithm implemented by Bresenham IP core.

All eight input parameters of Bresenham algorithm in Fig. 3 are the parameters of Bresenham IP core as well. That is, there is a unique memory-mapped register or flip-flop in the core with the same name that holds the respective parameter.

In order to start drawing a line, the Bresenham IP core driver running on CPU calls for brensenham_module with the initial values of these parameters. Specifically, in order to draw a line between (X1, Y1) and (X2, Y2), the driver initially sets these eight parameters as follows: major_y (1-bit): It is set true if Y-axis is the major axis which meets |Y2-Y1| > |X2-X1|. major_axis_first (32-bit): The lowest major axis component of either of the edge points, e.g. X1 if X is the major axis and X1<X2. major_axis_last (32-bit): The highest major axis component of either of the edge points. minor_axis_first (32-bit): Minor axis component of the edge point with the lowest major axis component between both edge points, e.g. Y1 if X is the major axis and X1<X2. error_major_shifted (32-bit): Two times of the change on major axis, e.g. (2×|X2-X1|), if X-axis is the major one. error_minor_shifted (32-bit): Two times of the change on minor axis. negative_direction (1-bit): It is set true if the minor axis component of the line being drawn is decreasing while the major axis component is going from major_axis_first to the major_axis_last. rgb (32-bit): Color of the line being drawn.

Once Bresenham function is started, it computes which pixel to fill in as follows: The while loop continues as long as major_axis_first is equal to or less than major_axis_last. Inside the while loop, fillPixel (rgb, x, y) function is called to fill the pixel in the frame buffer in (x, y) position with the color specified in rgb register. The major axis value of the next pixel (major_axis_first) is found by incrementing the current one by one. Whether the minor axis value (minor_axis_first) changes depends on the error value. If error is equal to or greater than zero, the minor axis value is incremented or decremented according to negative_direction flag. The error value is initially set to zero. Error is first decremented by error_major_shifted if it is equal to or greater than zero. Then, it is incremented by error_minor_shifted.

The implementation of bresenham_module is a finite state machine with three states: Initialization, FindPixelPosition and PixelFilling. In Initialization state, the module waits for the software driver to set 1-bit bre_go flag in a memory-mapped 8-bit flag register. Once this signal is set, the module reads the algorithm's parameters from the memory-mapped registers and goes to FindPixelPosition state. While the machine is not in Initialization state, it sets 1-bit bre_busy flag in the flag register so that the driver does not start a new line drawing operation. In FindPixelPosition state, the module runs Bresenham algorithm in Fig. 3 in order to compute a new pixel position, and goes to PixelFilling state. In PixelFilling state, if 1-bit dashed_line flag in the flag register is reset or set, Bresenham IP core draws a solid or dashed line, respectively. While drawing a dashed line, a 32-bit memory-mapped bre_ext_data register is used to adjust the interval between two dashes in a dashed line. The state machine goes to FindPixelPosition state from PixelFilling state if it is not the end of line, and to Initialization state, otherwise.

3.3. BitBLT IP Core

It is necessary to implement an exclusive peripheral for BitBLT operations because of their high cost. Fig. 4 shows the block diagram of BitBLT IP core. Similar to Bresenham IP core, BitBLT IP core is attached to PLB by means of PLB IPIF master-slave attachment modules (slave_read_module, slave_write_module, master_module). Thus, it can be master or slave. In addition to these modules, in Fig. 4, BitBLT IP core has a command_parser_module whose function is to provide an interface between WrFIFO (Write First-In-First-Out) Service of PLB IPIF and BitBLT_module. Finally, BitBLT IP core has a single 32-bit memory-mapped register, namely background_addr written by the core's driver.

WrFIFO (Write First-In-First-Out) Service of PLB IPIF basically provides a command FIFO buffer where BitBLT commands sent by CPU are temporarily kept. As long as there is an empty slot in WrFIFO, CPU can immediately insert a new BitBLT command into it whether or not BitBLT IP core is busy. After the insertion of a command, CPU can continue executing its other jobs. When WrFIFO is full, CPU is required to poll WrFIFO until there is an empty slot. The advantages of using WrFIFO are further emphasized in the design of polygon filling API with Scan Line method, which is built on top of BitBLT IP core.

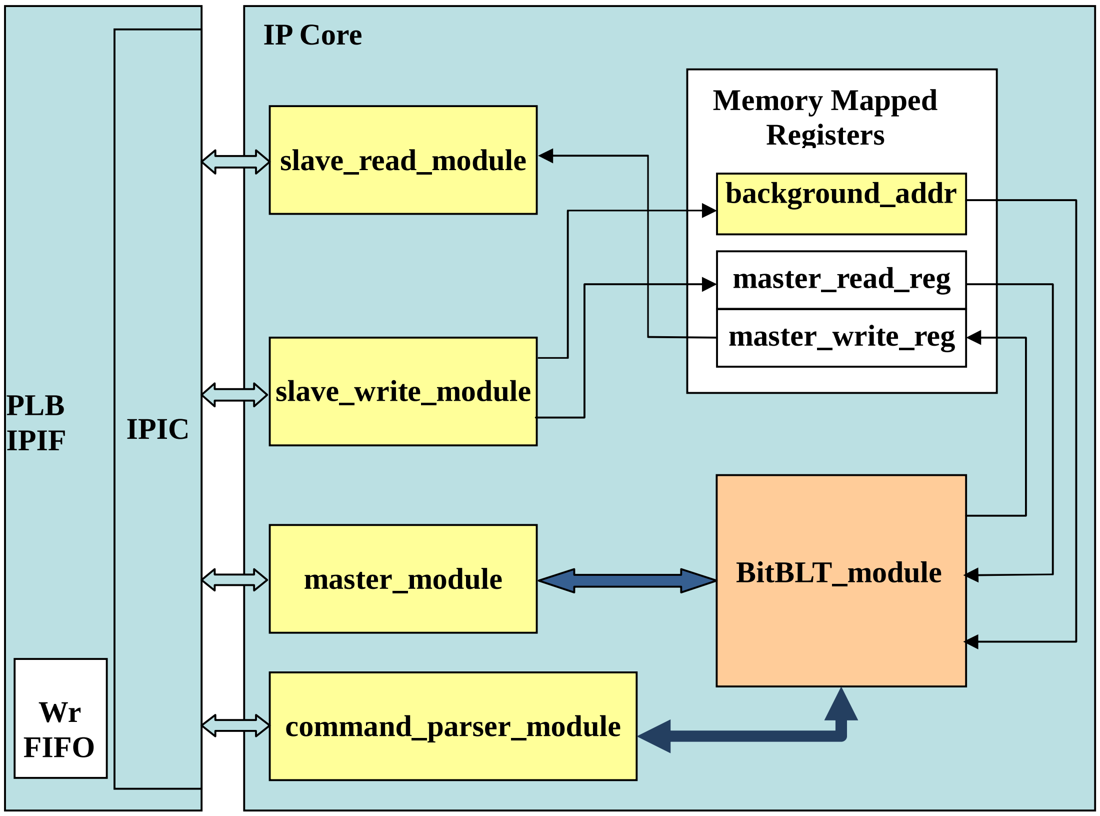


Figure 4. BitBLT IP core block diagram.

The BitBLT commands buffered in WrFIFO need to be fetched from there and dispatched to BitBLT_module one by one. This is the job of command_parser_module in Fig. 4: (1) It handles IPIC signals related with WrFIFO Service to provide a WrFIFO Service interface with BitBLT IP core. (2) It fetches BitBLT commands from the service one by one when WrFIFO is not empty. (3) It parses a fetched command; checks if it is a valid one; dispatches it to BitBLT_module by asserting blt_go signal if it is valid and BitBLT_module is not busy.

A command dispatched to BitBLT_module is composed of several fields some of which are as follows:

- argb (32-bit): Destination bitmap color, source bitmap color in source update, or color to transport or not transport in transparency
- bool_op (2-bit): 00: transport, 01: AND, 10: OR, 11: XOR
- transparency (2-bit): 00: not transparent, 01: transparent and pixels which are the same as argb will be transported, 10: transparent and pixels which are the same as argb will not be transported
- alpha (1-bit): 0: no alpha blending, 1: apply alpha blending
- corner_no (2-bit): BitBLT direction – 00: +X, –Y (→, ↑), 01: –X, –Y (←, ↑), 10: –X, +Y (←, ↓), 11: +X, +Y (→, ↓)
- source_update (2-bit): 00: source bitmap will not be updated, 01: source bitmap will be updated with argb, 10: source bitmap will be updated with background

BitBLT IP core designed in this study is a finite state machine with seven states that supports some generic and useful operations that can be simply grouped as follows (see [24] for a detailed explanation of the state machine and the list of BitBLT IP core operations):

- Filling destination with a specific color: The destination bitmap pixels are filled with a constant color.
- Write source to destination: The source bitmap pixel data are written to the destination bitmap with the options below:
  - Source update with a specific color: While writing to a destination bitmap, the source bitmap can be concurrently filled with a constant color.
  - Source update with a background bitmap: While writing to a destination bitmap, the source bitmap can be concurrently filled with another background bitmap.
  - Transparency: In the case that all pixels of an object’s bitmap do not represent the object, the transparency feature is a practical solution for transportation and Boolean operations, which eliminates the need for a mask. For example, suppose that there is a letter ‘A’ in a bitmap where the background color and the letter’s inside are green and the letter itself is red, and only the red pixels that represent the letter need to be transported. In order to accomplish this task, BitBLT IP core includes the transparency feature that makes it possible to command the core to transport or not to transport a specific color in the source bitmap. As a result, BitBLT IP core can be directed to transport only red pixels of letter ‘A’.
  - Corner: The starting corner and direction of BitBLT operation is important if source and destination bitmaps overlap. By choosing the right corner and direction, data loss due to overwriting the source bitmap can be prevented.
- Write source to destination with alpha blending: Alpha blending computation is made between each corresponding pixels of the source and destination bitmaps. The result is written to the destination bitmap. All four options listed above are also available for this operation.
- Write source to destination with Boolean operation: A Boolean operation is carried out between each corresponding pixels of the source and destination bitmaps. The result is written to the destination bitmap. All four options are possible for this operation as well.

It should be emphasizes that BitBLT IP Core is capable of performing BitBLT operations in 19 different types and 184 different configurations based on the command word.

4. GRAPHICS SYSTEM’S SOFTWARE DESIGN

In this study, drivers and graphics APIs are implemented in C programming language. The hierarchy of drivers and APIs is shown in Fig. 5. According to Fig. 5, IP cores are accessible through Xilinx IO library. This is due to the fact that IP core drivers use low-level functions in order to read from or write into the memory-mapped registers provided by Xilinx IO library.

4.1. Bresenham Driver

The driver of Bresenham IP core is composed of a header file and program file. The header file includes the core’s base address, the addresses of core’s memory-mapped registers (major_y, major_axis_first, etc.), the prototype of bresenhamLineDraw function, etc. The program file contains bresenhamLineDraw function itself. bresenhamLineDraw function has four input parameters: line edge points, line color, and solid/dashed line flag. When it is called, it first

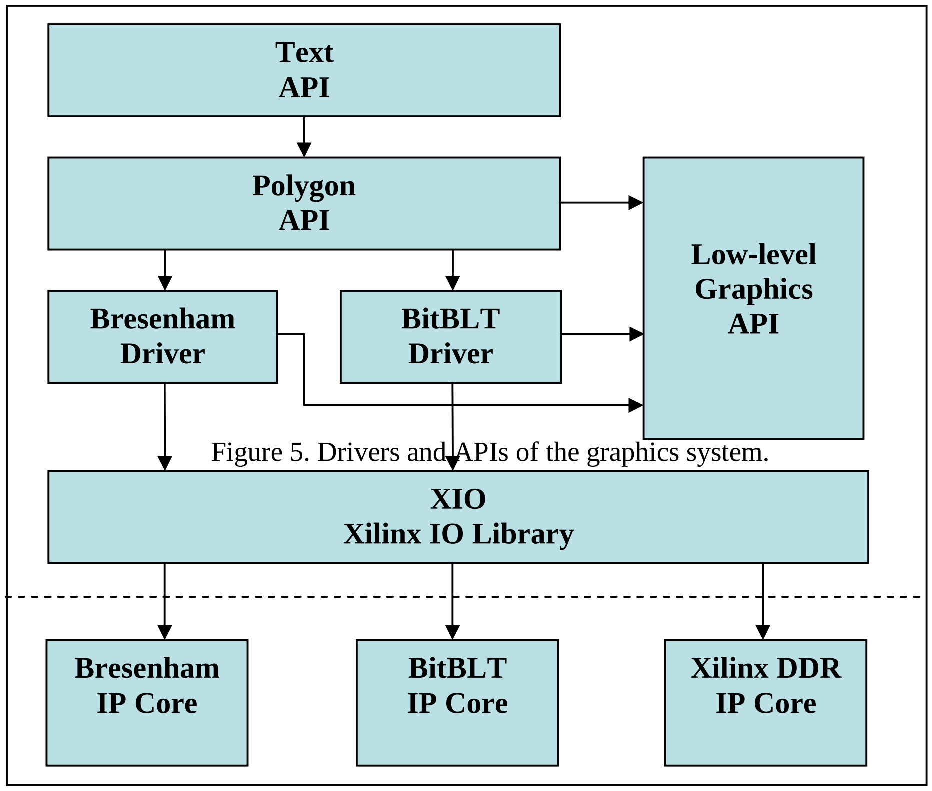


Figure 5. Drivers and APIs of the graphics system.

Figure 5. Drivers and APIs of the graphics system.

computes the input parameters of Bresenham IP Core with respect to the explanations in Section 3.2. After the computations, it controls the busy flag of the core by reading the related flag register and waits until it is ready. Then, it writes these parameters into the memory-mapped registers and sets bre_go flag by means of Xilinx XIO library functions.

4.2. BitBLT Driver

Similar to Bresenham IP core driver, a header file and program file constitute BitBLT driver. In the header file, there are the core's base address, the address of core's memory-mapped register (background_addr), bit-masks for the BitBLT command word, definition of the BitBLT command word, the prototype of bitBLT function, etc. In the program file, bitBLT function whose input parameters are the address of BitBLT IP core and BitBLT command word is found. This function writes the BitBLT command words into WrFIFO and the background image address into background_addr register. bitBLT function also uses Xilinx XIO library for accessing to WrFIFO and background_addr register.

4.3. Low-Level Graphics API

Low-level Graphics API has two header-program file pairs, namely graphCommon and coordSysXUP.

graphComman.h header file includes the definitions of RGB values of main colors, common graphics primitives (point, line, polygon, etc.), and prototypes of graphics utility functions. On the other hand, graphComman.c program file consists of the implementations of utility functions.

Some of these functions are getter, setter and builder functions of the structures defined in the header file. In addition, there are some other functions including pointRotate and premultipliedARGB. pointRotate function takes two points and an angle value as an input. One of the points is the axis and the other is the point that will be rotated around the axis up to the given angle. premultipliedARGB function takes an RGB value and the percentage of coverage as an input, and returns an ARGB value that comprises an alpha value and an RGB value multiplied by the coverage ratio.

coordSysXUP.h includes the address and resolution definitions of the frame buffer which is implemented by Xilinx VGA IP core. There is coordToAddr function implementation in coordSysXUP.c program file. This function takes a point as an input and returns the corresponding pixel address in the framebuffer.

4.4. Polygon API

Polygon API consists of polygon.c and polygon.h files. polygon.c file includes the implementations of polygonFill, polygonMove, polygonScale and polygonRotate functions, whereas polygon.h file consists of the prototypes of these functions. During the rasterization of a polygon, it is possible to observe all phases of a 2D rendering pipeline that includes geometric modeling, modeling transformations, viewing transformations, and rasterization.

The geometric modeling in Polygon API corresponds to the geometric definition of a polygon. In the API, a polygon is modeled by a set of sequential corner points with the help of polygon structure that is defined in graphCommon API. The polygon structure includes an integer edge count variable and an array of vertices. It should be noted that the polygon vertices in this phase are defined in the polygon’s self coordinate system whose origin is chosen to be one of the vertices.

Polygon API provides some useful functions, such as polygonMove, polygonScale, and polygonRotate, for the modeling transformation phase. polygonMove function takes a pointer to a polygon and a point to which the polygon will be moved. polygonScale function scales a polygon by multiplying the coordinates of all polygon’s vertices by a given scale value. polygonRotate function rotates every corner point of a polygon around a given axis point up to a given angle value by means of pointRotate function of Low-level Graphics API.

polygonFill function implements viewing transformation and rasterization phases. First, polygonFill function finds out all horizontal line segments that should stay inside of the polygon being drawn and need to be filled. Then, for every such horizontal line segment, polygonFill function puts the related command words into WrFIFO of BitBLT IP core. In the BitBLT command word, the start address of a horizontal line segment in framebuffer is calculated by means of coordToAddr function of Low-level Graphics API. Using coordToAddr function corresponds to the viewing transformation. Finally, BitBLT IP core fills a line segment by simply writing a given color value into the part of framebuffer that corresponds to this line segment. It should be noted that CPU IP Core can enqueue a new command word into WrFIFO while BitBLT IP Core is filling a line. As a result, the waste of CPU time during the polygon filling is minimized.

4.5. Text API

Text API is designed to show that all parts of the graphics system are functioning as required. According to Fig. 5, Text API is the highest level API among available APIs and it is situated on top of Polygon API.

The prototype and implementation of writeText function that is used to print texts to framebuffer are found in Polygon API. writeText function's parameters are the location where the text will be printed, character string, text color, and text size. There are also other functions that are not a part of API's interfaces, but they are used by writeText function. For example, charPosition function is used to calculate the locations of the successive characters.

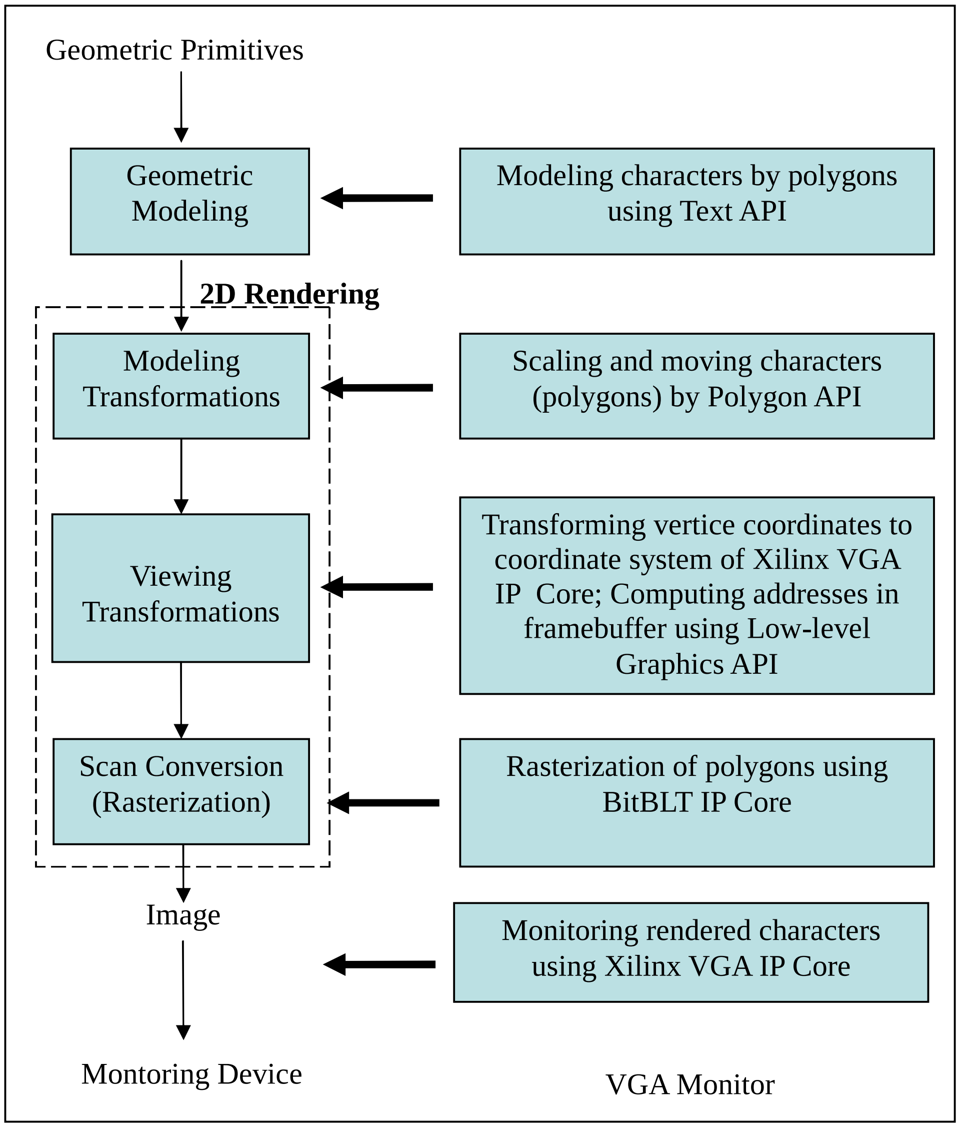

Geometric Primitives
Geometric Modeling
Modeling characters by polygons using Text API
2D Rendering
Modeling Transformations
Scaling and moving characters (polygons) by Polygon API
Viewing Transformations
Transforming vertice coordinates to coordinate system of Xilinx VGA IP Core; Computing addresses in framebuffer using Low-level Graphics API
Scan Conversion (Rasterization)
Rasterization of polygons using BitBLT IP Core
Image
Monitoring rendered characters using Xilinx VGA IP Core
Montoring Device
VGA Monitor

Figure 6. A comparison of character rendering in the proposed graphics system and 2D rendering pipeline.

In Text API, all characters are modeled by polygons. Each character is always defined by a set of 10×10 cells where the coordinates of all vertices are given based on the coordinates of top-left corner (a,b) and each cell is assumed to have the thickness value of k. In Text API, all characters are initially defined for (a,b) = (0,0) and k=1. By means of polygonMove and polygonScale functions in Polygon API, it is possible to change these three parameters (a, b, and k) as required, which makes it possible to draw any character on the display at any given location and scale.

5. SYSTEM DEMONSTRATION

The graphics system designed includes all phases of 2D rendering pipeline as shown in Fig. 6. In the geometric modeling phase, polygons and characters are modeled by points and polygons, respectively. In the modeling transformations phase, there are polygon scaling and moving

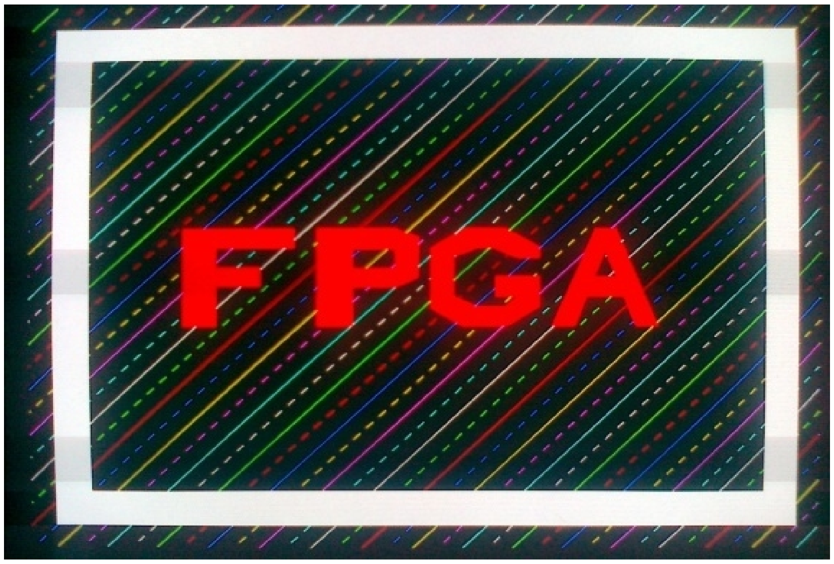


Figure 7. An example image of monitor output rendered fully by the graphics system implemented in Virtex-II FPGA.

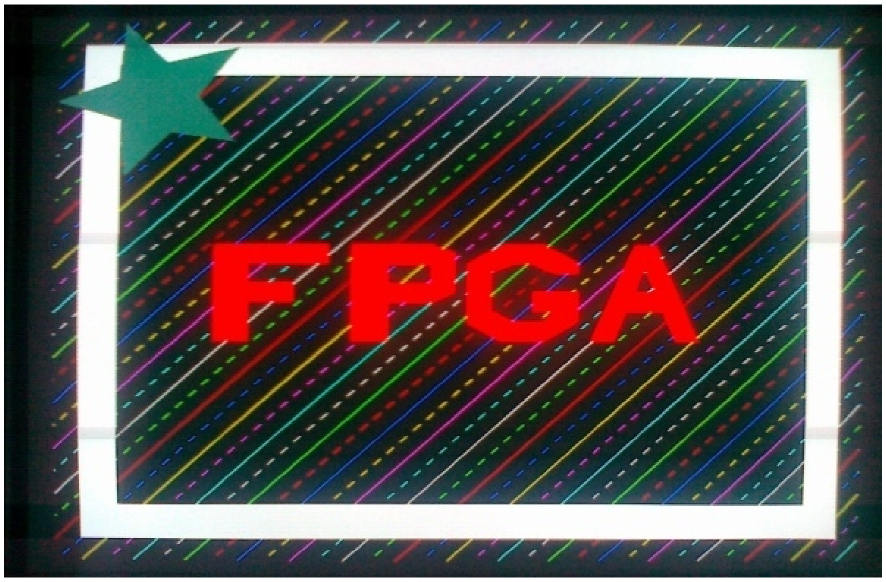


Figure 8. An opaque star is drawn on the background image.

functions both of which are defined in Polygon API. Low-level graphics API that transforms the coordinates of polygon vertices into the screen coordinate system constitute the viewing transformation phase. For the scan conversion in which polygons are drawn and filled, BitBLT IP core and Bresenham IP core are used. Finally, the monitoring phase is implemented with the help of Xilinx's VGA IP core. The graphics system was tested by connecting a monitor to VGA output of Virtex-II development board. Fig. 7, 8, and 9 show the examples of monitor images rendered fully in Virtex-II FPGA chip on the development board.

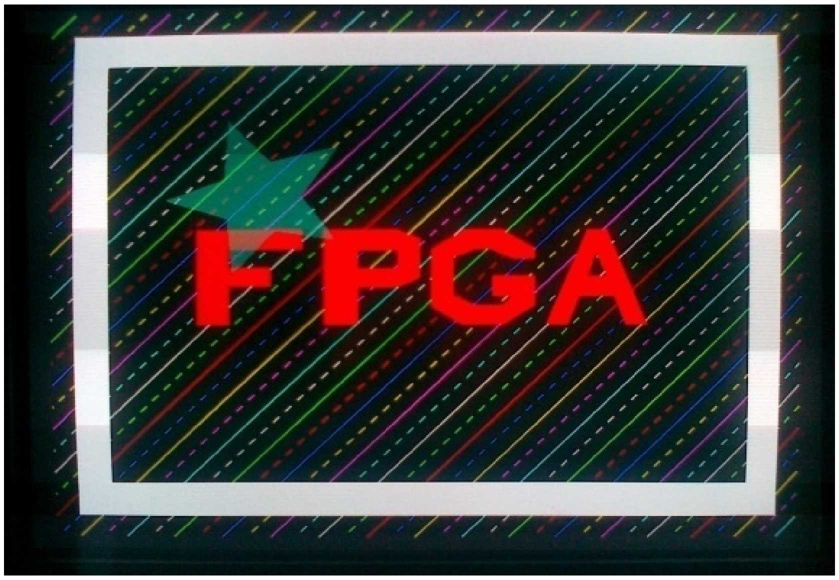


Figure 9. Moving and alpha blending the star.

In Figure 7, Bresenham IP core and its driver draw continuous and dashed lines; BitBLT IP core and its driver fill white frame; Text API (indirectly Polygon API and BitBLT IP core) is used to print text "FPGA".

The monitor image in Fig. 7 is saved as a background image in another part of the framebuffer. Then, as shown in Fig. 8, an opaque star is drawn on the background image by means of Polygon API. The opaque star is drawn without alpha blending option.

Fig. 9 shows that the opaque star in Fig. 8 is moved to another location in the screen while it is made transparent by alpha blending with the background image. This is achieved by a single BitBLT command. BitBLT IP core updates the star's old location with the background image, moves the star to its new location while it alpha blends the star with the destination's background image. Furthermore, the transparency feature of BitBLT IP core is also used so that the old location background image that is not the same as the star's color is not moved to the new location.

According to the synthesis result of XST (Xilinx Synthesis Tool), the current BitBLT IP core design requires 3270 slices, 2922 slice flip-flops, and 5701 four-input LUTs if PLB IPIF entity that comes with EDK is instantiated; 1752 slices, 1486 slice flip-flops, and 3069 four-input LUTs, otherwise. In order to give an idea of timing performance of BitBLT IP core, one of the BitBLT's longest operation types, which is writing source to destination with alpha blending and with source updating with background image, is considered. This operation can be started with only one BitBLT IP command word. BitBLT IP core's state machine loops in states for every pixel as follows: read from source, read from destination, calculate and write to destination, read from background image, write to source, calculate next pixel position. One single beat master read operation requires 17 clock cycles and one single beat master write operation requires 19 clock cycles according to PLB IPIF timing diagram [21]. BitBLT states which do not interact with PLB cost only one clock cycle. Therefore, aforesaid BitBLT operation needs 91 clock cycles for every pixel. This means 910 nanoseconds (ns) for every pixel with 100 MHz system clock. On the other hand, filling a destination with a specific color takes only 200 ns. The movement of the opaque star to its final destination shown in Fig.9 with alpha blending and source update option took approximately 7 ms.

Again according to XST, the current design of Bresenham IP core requires 2482 slices, 2415 slice flip-flops, and 4383 four-input LUTs with the instantiation of PLB IPIF entity. Otherwise, it requires 1042 slices, 1095 slice flip-flops, and 1850 four-input LUTs. Bresenham IP core can rasterize a line that of 500 pixels in 0,1 ms.

6. Conclusions

In this study, a 2D graphics system composed of a CPU IP core and a few peripheral IP cores is implemented on FPGA. The design work is split into two main parts as hardware and software design. In the hardware design, first, two graphics IP cores, namely Bresenham and BitBLT are designed. Then, an interface logic is developed based on PLB IPIF so that the graphics IP cores can be connected to PLB bus and they can be controlled by their respective drivers. Finally, a PowerPC CPU core, Bresenham and BitBLT graphics IP cores, and two other peripheral IP cores are brought together around a PLB bus to complete the hardware design.

During the software design, the drivers of in-house IP cores and some graphic APIs are developed in C programming language. With API functions, the graphics creation phases before rasterization are implemented. On the other hand, the drivers interact with the related IP cores to realize rasterization and after-rasterization phases.

As a future work, other graphics IP cores with additional features such as drawing ellipse and other geometric primitives, antialiasing and texturing can be designed and easily added into the system. Other color formats (RGB565, ARGB4444, etc.) can be supported by BitBLT and monitoring IP cores. Furthermore, other useful display IP cores can be adapted for the system. On the software side, new APIs and user interface frameworks can be developed for embedded systems. All of these improvements are made possible by the flexible and extendable hardware and software design of the graphics system proposed.

References

[1] W. Wolf. High-Performance Embedded Computing, Architectures, Applications, and Methodologies. Morgan Kaufmann, 2007.

[2] G. De Micheli, R. K. Gupta, "Hardware/software co-design," Proceedings of IEEE, 85(3), 1997.

[3] S. Mu, C. Wang, M. Liu, D. Li, M. Zhu, X. Chen, X. Xie, Y. Deng "Evaluating the potential of graphics processors for high performance embedded computing, " *DATE*, pp. 1-6, 2011.

[4] S. Singh, P. Bellec, "Virtual hardware for graphics applications using FPGAs," *IEEE Workshop on FPGAs for Custom Computing Machines*, 1994.

[5] Altera Company (www.altera.com), AN371: Automotive Graphics System Reference Design, 2004.

[6] L. Gray, R. Woodson, A. Chau, S. Retzlaff, "Graphics for the long term: an FPGA-based GPU," *VMEbus Systems*, 2005.

[7] Altera Company (www.altera.com), Using FPGAs to Render Graphics and Drive LCD Interfaces, 2009.

[8] N. Knutsson. An FPGA-based 3D Graphics System. Linköping Institute of Technology, Master of Science Thesis, 2005.

[9] H. Holten-Lund, “An embedded 3D graphics core for FPGA-based system-on-chip applications,” *FPGAworld Conference*, 2005.

[10] J. R. Warner. Real Time 3-D Graphics Processing Hardware Design Using Field-programmable Gate Arrays. University of Pittsburgh, Master of Science Thesis, 2008.

[11] LogicBrics Company (http://www.logicbricks.com), logiBITBLT IP Core DataSheet.

[12] Imagination Company (http://www.imgtec.com), POWERVR MBX GPU IP Core.

[13] TES Electronic Solutions (www.tes-dst.com), D/AVE 2D Data Sheet.

[14] R. Moller, “State-of-the-Art 3D graphics for embedded systems,” in *Proc. of the 6th International Caribbean Conference on Devices, Circuits and Systems*, pp. 39-43, 2006.

[15] S.-H. Chen, H.-M. Lin, H.-W. Wei, Y.-C. Chen, C.-T. Huang, Y.-C. Chung, “Hardware/software co-designed accelerator vector graphics applications,” IEEE Symposium on Application Specific Processors, pp. 108-114, 2011.

[16] M. Dutton. Flexible Architecture Methods For Graphics Processing. Georgia Institute of Technology, PhD Thesis, 2011.

[17] F. Guo, W. Wan, X. Zhang, X. Zhou, “Design of test platform for 3D graphics pipeline based on microblaze,” International Conference on Audio, Language and Image Processing, pp. 392-396, 2012.

[18] F. Guo, W. Wan, X. Zhang, X. Zhou, “Research of graphics acceleration based on embedded system,” International Conference on Audio, Language and Image Processing, pp. 1120-1124, 2012.

[19] D. Hearn, M. P. Baker. Computer Graphics C Version, 2nd Edition. Prentice Hall, 1997.

[20] T. Porter and T. Duff, “Compositing digital images”, Computer Graphics, 18 (3), pp. 253–259, 1984.

[21] A. R. Smith. Image Compositing Fundamentals. Microsoft Corporation, 1995.

[22] IBM (www.ibm.com), CoreConect Bus Specifications.

[23] Xilinx (www.xilinx.com), PLB IPIF (v2.2a) Product Specification.

[24] K. S. Ay. FPGA Based Graphical System Design for Embedded Systems. Anadolu University, Master of Science Thesis, 2010.