2D Graphics Engine:
Group Report

ECE532: Digital Hardware

Hafiz Noordin (990006747)
Mohamad Ayad (993069353)
Porya Rajabi (992464962)

April 2, 2007

Contents

1. Overview

1.1 Objective

A graphics system consists of several levels of hardware and software working together to make a computer capable of displaying visual information on a monitor. The primary engine behind the graphics system is the graphics controller, which consists of the custom hardware processor that performs all necessary mathematical operations required to generate pixel data (RGB colour), which is then stored in a region of memory called the frame buffer. A display controller then reads the frame buffer and converts the RGB data to pixels on-screen. A software driver is also necessary to stimulate the hardware and provide an API for developers to create graphics applications.

Modern graphics controllers are usually implemented as ASIC's (Application Specific Integrated Circuits), primarily due to their high performance requirements, especially for intensive 3D applications. The goal of our project was to implement a graphics system on an FPGA. In particular, we designed and implemented the necessary hardware and software components to perform basic 2D operations.

1.2 Goals

- Research components and algorithms to perform basic 2D operations
- Design Xilinx FPGA-based system incorporating 2D engine, display controller, memory, microprocessor
- Design and implement digital circuits for 2D engine operations
- Implement software API
- Develop software application to demonstrate functionality of system
- Desired operations:
    - Draw pixel
    - Blit (fill a rectangular region on the screen)
    - Draw line
    - Draw character
    - Bitmap (read raw bitmap information from memory and display on screen)

1.3 System Overview

Figure 1 illustrates the System Block Diagram for the 2D graphics system. We used the Multimedia board based on the Virtex-II XC2V2000-FF896 FPGA. The components of the system are divided based on their physical location/implementation: inside the FPGA, on the Multimedia board, and external to the board.

The components of our project designed from scratch are shown in green, including:

- 2D Graphics Engine: hardware block implemented as an OPB slave (coded in Verilog)

- Gfx2D API: software IP stored on ZBT RAM 1 via EMC controller (coded in C)
- Demo application: software stored on ZBT RAM 1 via EMC controller (coded in C)

The components shown in red were part of an IP core that was imported and implemented based on the Bit Mapped Mode SVGA example provided by Xilinx. This module provides an interface to write pixel data to a ZBT RAM, and then transmits the contents of ZBT RAM 2 (frame buffer) to the VGA DAC.

The system works by first issuing a command by software using the Gfx2D API. A call to an appropriate C function in the API will issue a command to the 2D Graphics Engine via a register write on the OPB bus. Note that prior to this step, the OPB bus is used to read the instruction in the user's program code from ZBT RAM 1. The graphics engine then processes the command and generates pixel data based on the desired function. For example, a command to draw a green line from (0,0) to (100, 200) will cause the graphics engine to determine all pixels necessary to draw the appropriate line with an RGB value representing green. The pixel data is written to ZBT RAM 2 via the ZBT interface shown, and stored to the area of memory that represents the contents of what is being displayed to the screen. The display controller constantly reads the contents of the frame buffer from ZBT RAM 2 and updates the RGB data lines that the SVGA DAC (Digital to Analog Converter) uses to display pixels on the monitor.

Each of the components in the system will be further described in Section 3 below.

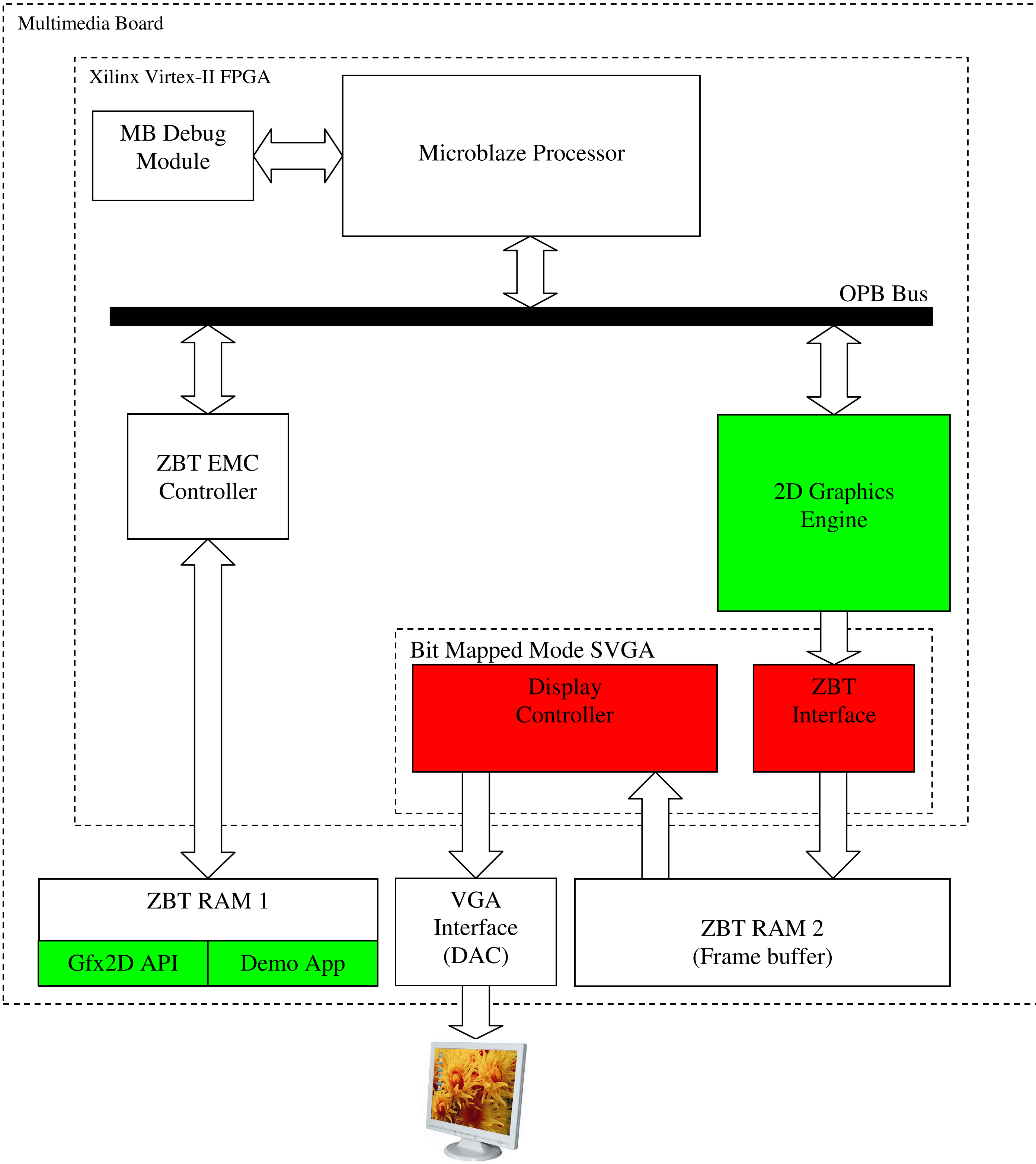


Figure 1: System Block Diagram

2. Outcome

2.1 Results

All goals were accomplished successfully. The only modification to our initial goals was implementing the bitmap feature in software rather than hardware. This decision was made due to the time constraints. The hardware version would have required more time to implement due to the extra complexity in implementing a custom ZBT interface to both read and write to the memory, and more importantly be able to coordinate writing a large amount of bitmap data. The software version was much more straightforward since XPS supports statically linking and loading the user program code (including the data variable that stores the bitmap data) to a ZBT RAM via the ZBT EMC controller.

The 2D engine functions correctly. All 2D operations were tested and demonstrated successfully, and the overall system was able to run from software. As described, the program was setup to be stored on and read from ZBT RAM 1, which allowed bitmap files to be statically linked and loaded to our program.

The software API was developed to be capable of running a low-level C function corresponding to each of the 4 basic 2D graphics operations implemented in hardware: draw pixel, blit, draw line, and draw character. These low-level C functions issue register writes to the 2D engine in order to pass the function parameters. The decoder in the 2D engine reads the op-code and triggers the corresponding hardware module. Each module stores its resultant pixel data to a FIFO. An arbiter at the bottom of the pipeline reads from these FIFO's and writes the pixel data to the frame buffer (ZBT RAM 2). We implemented the following low-level functions:

- **lineOp**: Draw a line on screen using two points and a colour.
- **blitOp**: Fill an area defined by a rectangle with a specific colour.
- **charOp**: Draw a character on screen at a given coordinate.
- **pixelOp**: Draw 1 pixel on screen at given coordinate with specified colour.

The following high-level functions were implemented in C code as part of the Gfx2D API. They perform operations to draw more advanced primitives by manipulating the low level functions. Each function is passed parameters about the colour it should use to draw, as well as the coordinates to place the object on screen.

- **clearScreen:** Fill the entire screen with a single colour.
- **drawRect**: Draw the a rectangle (unfilled).
- **fillRect:** Draw and fill a rectangle.
- **drawSquare**: Draw the a square (unfilled).
- **fillSquare**: Draw and fill a square.
- **drawTriangle**: Draw a triangle (unfilled).
- **drawStar**: Draw a 4-point star (unfilled)
- **rotateSquare90**: Draws and rotates a square 90 degrees.
- **rotateStar90**: Draws and rotates a star 90 degrees.
- **drawString**: Write given string on screen (using charOp).
- **ppmOp**: Draw a ppm (raw RGB) picture on screen ("bitmap mode")

A software application was developed to demonstrate all above functions.

2.2 Future Work and Improvement

The following are some proposed features for future work:

- **2D Operations**: The graphics engine was designed to be easily expanded for more custom 2D operations. We could easily implement additional algorithms such as drawing a circle, curved line drawing, or textures.

- **Image Manipulation**: Functionality can be added to skew, stretch, invert, or rotate an image on screen.

- **Advanced Frame Buffer**: Double buffering can be added to allow the engine to work on one buffer without it showing up on screen. Once the engine is done, then the buffers can be swapped to instantly update the frame buffer. Furthermore, the size of the frame buffer can be increased to support higher bits per channel or higher resolutions.

- **Effects**: Image effects like shading, alpha blending, or blurring can be added.

- **Media**: Support for image/movie files can be added. This can be something as simple as parsing a bitmap header. For full scale projects like a JPEG or MPEG decoder, our project can be used as a very good starting point.

- **Advanced Software API**: The API can be enhanced to be more object oriented, so that to allow more powerful operations. For instance, a square can be drawn on screen, and an object-based API can be used to move the square around or change its colour.

3. Description of Blocks

As previously illustrated in Figure 1, the components of the system are divided based on their physical location/implementation: inside the FPGA, on the Multimedia board, and external to the board.

3.1 Components inside FPGA

The hardware implemented on the FPGA was designed using the Xilinx XPS tool, version 8.2.02i. The Base System Builder (BSB) Wizard was used to instantiate the necessary cores imported from the IP catalog, including the following major blocks:

- Microblaze processor
- OPB bus

- Local memory buses
- Microblaze debug module
- Digital clock manager (DCM)
- ZBT external memory contoller (EMC)

We then developed two additional IP cores: the graphics 2D engine, and the display controller. As previously mentioned, the graphics 2D engine was a custom design developed from scratch by our team, while the display controller was imported and implemented based on the Bit Mapped Mode SVGA example provided by Xilinx.

The following sections describe all IP blocks and custom hardware implemented on the FPGA. Except where explicitly stated, all modules were instantiated once.

3.1.1 Microblaze processor

IP Name: microblaze
Version: 5.00.c
Instance: microblaze_0
Source: XPS IP catalog

The Microblaze processor is used to read program instructions from the ZBT memory (ZBT RAM 1) via the ZBT EMC controller during program execution (using a generated linker script). The instruction/data is processed, and once a graphics operation is reached the processor executes register write instructions via the OPB bus to the 2D Graphics engine. The 2D engine then decodes the op code and triggers the appropriate graphics operation.

IP also instantiated by BSB with Microblaze:

- Instruction/Data Local Memory Busses (ilmb/dlmb – lmb_v10, 1.00.a)
- Instruction/Data Local Memory Bus Controllers (ilmb_cntlr/dlmb_cntlr – lmb_bram_if_cntlr, 2.00.a)
- Block RAM (lmb_bram – bram_block, 1.00.a)

3.1.2 OPB bus

IP Name: opb_v20
Version: 1.10.c
Instance: mb_opb
Source: XPS IP catalog

The OPB bus is the primary communication link in the system. It is used by the Microblaze processor to:

- Load executable to ZBT memory by writing to ZBT EMC controller
- Read instructions from ZBT memory when program execution begins
- Send register writes to graphics engine when graphics operation is requested

- Read graphics engine registers for status signals

3.1.3 Microblaze Debug Module

IP Name: opb_mdm
Instance: debug_module
Version: 2.00.a
Source: XPS IP catalog

Required to support JTAG-based debug tools, i.e. XMD.

3.1.4 OPB External Memory Controller (for ZBT EMC)

IP Name: opb_emc
Instance: ZBT_512Kx32
Version: 2.00.a
Source: XPS IP catalog

The OPB external memory controller is an OPB slave device that was instantiated to read from and write to an external ZBT memory bank. Our bitmap function had to be able to store the raw RGB data of a picture in memory and since the BRAM did not have enough capacity, a 2nd ZBT had to be added to the project. The EMC was added using the Base System Builder in XPS. After BSB instantiated the core, several LOC constraints were added to the system.ucf file according to module m08. The ZBT RAM that was connected to the EMC stored the entire program executable, and the program would statically link the ppm data as an initialized variable. The executable was loaded in XMD via the JTAG/MDM interface and the generated linker script. The Microblaze processor would then begin program execution at the ZBT RAM address.

The changes to this module were guided by module m08:
http://www.eecg.toronto.edu/~pc/courses/edk/modules/6.3/m08.pdf

IP also instantiated by BSB with opb_emc:

- Utility Bus Split (ZBT_512Kx32_util_bus_split_1 – util_bus_split, 1.00.a)

3.1.5 Digital Clock Manager (DCM)

IP Name: dcm_module
Instance: dcm_0
dcm_module_0
Version: 2.00.a
Source: XPS IP catalog

The first DCM module was required to generate the digital clock signals that were used by all FPGA-based blocks. The external clock would feed in and the DLL would generate the desired 27 MHz clock that would be used throughout the FPGA. We implemented our system as a single clock domain. The same clock was also buffered through into the pixel clock for the VGA controller, as well as the memory clock for the frame buffer ZBT RAM.

The second DCM module was added when the 2nd ZBT RAM was instantiated. This DLL used the system clock generated by the first DCM module, and fed its output clock signal directly to the 2nd ZBT RAM.

The changes to this module were guided by module m08:
http://www.eecg.toronto.edu/~pc/courses/edk/modules/6.3/m08.pdf

IP also instantiated with dcm_module_0:

- Clock Align module (clk_align_0 – clk_align, 1.00.a)
  This core was added according to the instructions in module m08. Since a DCM needs time to lock and synchronize the clocks, a measure has to be taken to keep the components of the system in a reset state during the synchronization. The Clock Align module takes care of this.

3.1.6 Bit Mapped Mode SVGA – Display Controller

IP Name: display
Instance: display_0
Version: 1.00.a
Source: Code imported from Xilinx web site (Verilog) (http://www.xilinx.com/products/boards/multimedia/docs/examples/BM_MODE_SVGA.zip). pcore created using ISE and Create/Import Peripheral Wizard in XPS.

Figure 2 below illustrated the SVGA controller. The "user" is given access to read and/or write to the ZBT RAM via the ZBT controller. Based on the signals generated by the SVGA Timing Generator, data was read from the ZBT RAM and displayed to the screen by driving the Video DAC's RGB lines.

This was the only "imported" code, i.e we obtained the verilog code but had to manually create a pcore using both ISE and the Create/Import Peripheral Wizard in XPS. The system.mhs file was manually edited to connect the display controller directly to the 2D graphics engine. The system.ucf file was manually edited to add the pinout for the VGA DAC signals and the external ZBT interface for the frame buffer RAM.

The function of each verilog file in this pcore is described directly in the source code. Please refer to the header comment for each file:

- display.v

- ADDR_BUS_INTERFACE.v
- CTRL_BUS_INTERFACE.v
- DATA_BUS_INTERFACE.v
- DRIVE_DAC_DATA.v
- MEMORY_CTRL.v
- PIPELINES.v
- SVGA_TIMING_GENERATION.v
- ZBT_CONTROL.v

The source code was modified to hard-code the controller to operate at a resolution of 640x480 @ 60Hz, and a pixel clock of 25.175 MHz, thus the 27 MHz system clock was sufficient for the pixel clock. This modification was done in SVGA_TIMING_GENERATION.v. Also the Clock Mux was removed and rewired such that both the ZBT controller and ZBT RAM used the pixel clock rather than inputting a separate user_clock.

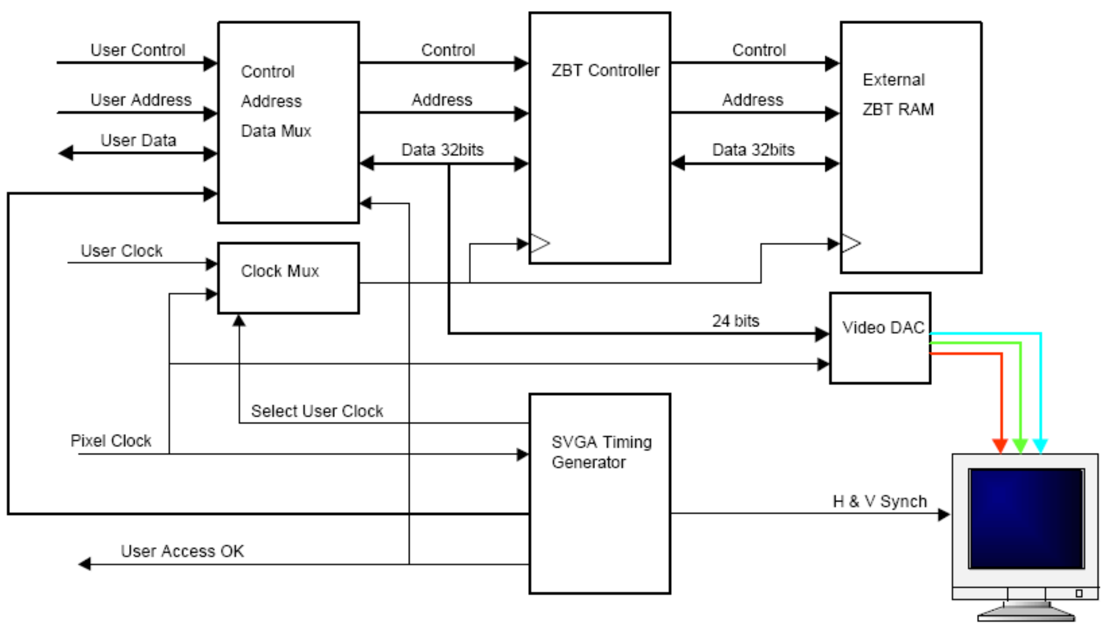


Figure 2: Block Diagram of the Bit Mapped Mode SVGA Controller (Source: http://www.xilinx.com/products/boards/multimedia/docs/examples/bm_mode_svga.pdf)

3.1.7 2D Graphics Engine

IP Name: gfx2d
Instance: gfx2d_0
Version: 1.00.a
Source: Custom design (Verilog)

Figure 3 below illustrates the 2D Graphics pipeline.

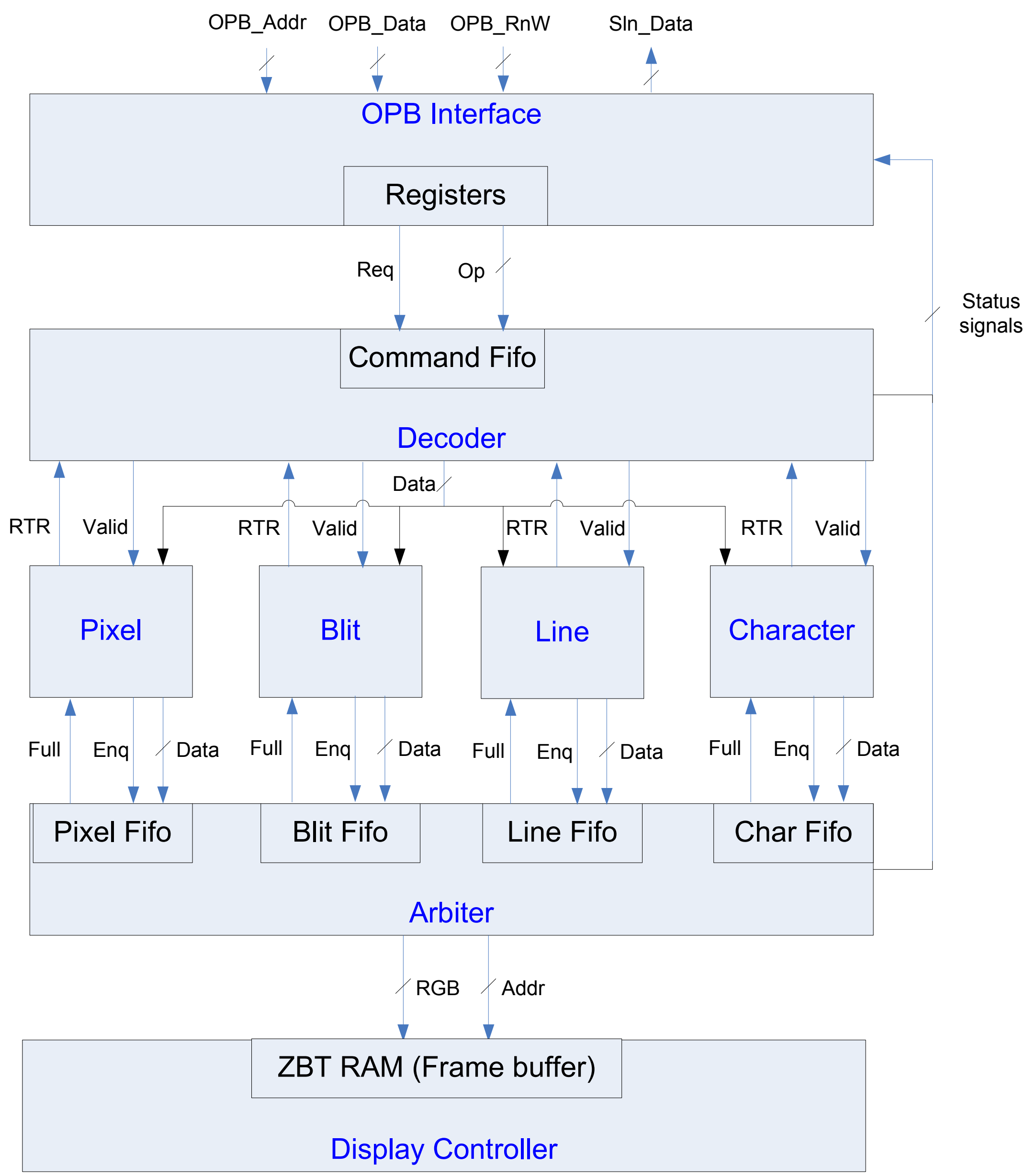


Figure 3: 2D Graphics Engine - Block Diagram

The 2D Graphics Engine is the heart of our system. The OPB interface accepts register read/write requests from the Microblaze processor in order to stimulate the engine. The input operation and data are written to registers. When the request bit is written to via an appropriate register write, the op and data are stored into a Command FIFO stored in the Decoder block. The Decoder then dequeues from the FIFO when the 2D operation modules (Pixel, Blit, Line, Character) are available to accept a request. The op code is decoded and the valid line associated with the target module is raised, and the data bus is driven with the input data dequeued from the FIFO. The target 2D module begins operating on the data and lowers its RTR (ready to receive) until it has completed.

During its operation, the 2D module stores generated pixel data into an output FIFO. The full signal indicates if the FIFO is full, at which point the 2D module would stall until space is available to store pixel data. The Arbiter then alternates between all FIFO's and dequeues pixel data. This data is then written to frame buffer memory by interfacing with the display controller module.

A pcore to encapsulate the 2D graphics engine was developed by manually creating MPD, PAO, and BBD files. The module was designed as an OPB slave with a 256 byte address space.

3.1.7.1 Parameters

The MPD file for this module defined the following parameters:

Table 1: MPD Parameters

Parameter Name	Feature Description	Allowable Values	Default	Type
C_BASEADDR	Base address in OPB space	Same as primitive	FFFF_FF00	Integer
C_HIGHADDR	High address in OPB space	Same as primitive	FFFF_FFFF	Integer
C_OPB_AWIDTH	Width of OPB address bus	Same as primitive		Integer
C_OPB_DWIDTH	Width of OPB data bus	Same as primitive		Integer

3.1.7.2 Register Specification

1) C_BASEADDR: Status register – contains the request, op, and status bits.

Table 2: Status Register

Bits	Name	Description	Reset Value
2:0	Op code	Current op code. Written by user.	0
3	Request	Request bit set by user. When user writes a 1, the request signal is triggered for 1 clock cycle and is then deasserted.	0
4	blit_rtr	Blit module is ready.	1
5	line_rtr	Line module is ready.	1
6	char_rtr	Character module is ready	1
7	gfx2d_rtr	Graphics 2D Engine is ready.	1
8	arb_mem_rtr	ZBT Frame buffer – user access is okay. This value is not set by the graphics engine thus it's reset value is unknown.	X
9	arb_blit_full	Blit output FIFO is full.	0
10	arb_line_full	Line output FIFO is full.	0
11	arb_char_full	Character output FIFO is full.	0
12	arb_pixel_full	Pixel output FIFO is full.	0

31:13	Reserved	Not used	0

2) C_BASEADDR + 4 → C_BASEADDR + 20: Input data – data registers set by user.

Table 3: Input Data Registers

Bits	Name	Description	Reset Value
31:0	Input_data1	Current input data (word 1)	0
63:32	Input_data2	Current input data (word 2)	0
95:64	Input_data3	Current input data (word 3)	0
127:96	Input_data4	Current input data (word 4)	0
159:128	Input_data5	Current input data (word 5)	0

3) C_BASEADDR + 24: Debug fifo data – current debug FIFO data if debug FIFO not empty.

Table 4: Debug FIFO Register

Bits	Name	Description	Reset Value
31:0	Input_data1	Current debug FIFO output. If empty, defaults to reset value.	DEAD_BEEF

3.1.7.3 Description of Blocks

Top-level module
File: gfx2d.v
Description: top-level module which encapsulates all other modules in design. Also generates signals for pixel operation, as well as memory interface.

OPB Interface
File: OpbInterface.v
Description: OPB slave interface for register reads/writes. Stores registers to accept request operations and data. Adapted from module m05 – OpbInterface.v. A small FSM is used to maintain the block in its idle state after reset. When a register read request arrives, the FSM drives the OPB bus with the requested register data. When a register write request arrives, the FSM loads the respective register with the data from the OPB bus. In both cases the FSM goes to a state in which the OPB bus is driven back to zero, and return to its idle state.

Decoder
File: Decoder.v
Description: The Decoder passes data from the Command FIFO to the operations based on the op code of the data. It first dequeues the Command FIFO, decodes the op code from the data, and then raises the valid signal of one of the 4 operations. Figure 4 shows the digital circuit used to design the Decoder in Verilog. The circuit consists of combinational logic to dequeue the FIFO and to validate the operations. The Decoder stalls until all operations are ready-to-receive (RTR). Once all operations are RTR, the FIFO is dequeued and the data is passed to the Decoder. The data's least significant 160

bits is placed on a data bus (D_out), to be read by all operations, and the most significant 3 bits are compared with the designed OP code to validate one of the four operations.

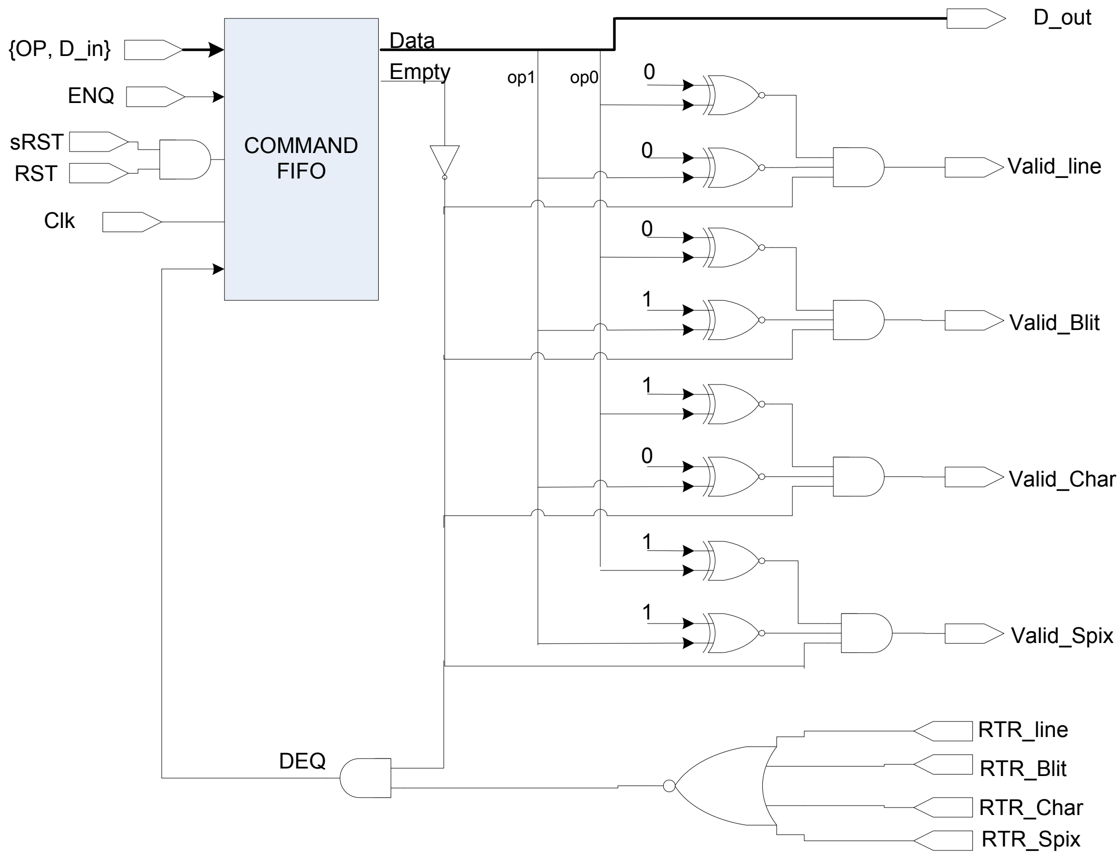


Figure 4: Decoder architecture

The data width, op code for each algorithm, and the op code width are written as parameters in the Verilog code, so can be easily modified if more operations are added to the system. The current op codes are:

Table 5: Op Codes

OP code (b_1,b_0)	Operation
000	Line
001	Blit
010	Character
011	Set pixel
100	Debug

<u>Blit / Pixel Drawing</u>
File: Blit.v, gfx2d.v
Description: Pixel drawing basically means there is no processing done to the input data. The input coordinates and RGB are sent directly to the pixel output fifo. Blit requires the use of two counters to keep track of the current x and y coordinates. Based on the start (x0,y0) and end (x1,y1) coordinates, the logic increments the x counter until x = x1, at

which points the y counter is incremented and x is reset to x0. This process continues until (x,y) = (x1,y1). On each clock cycle, if the output FIFO is not full, the pixel data is written and the counter(s) are incremented. If not, the counters do not change (stall). Figure 5 shows a high level diagram of the blit module.

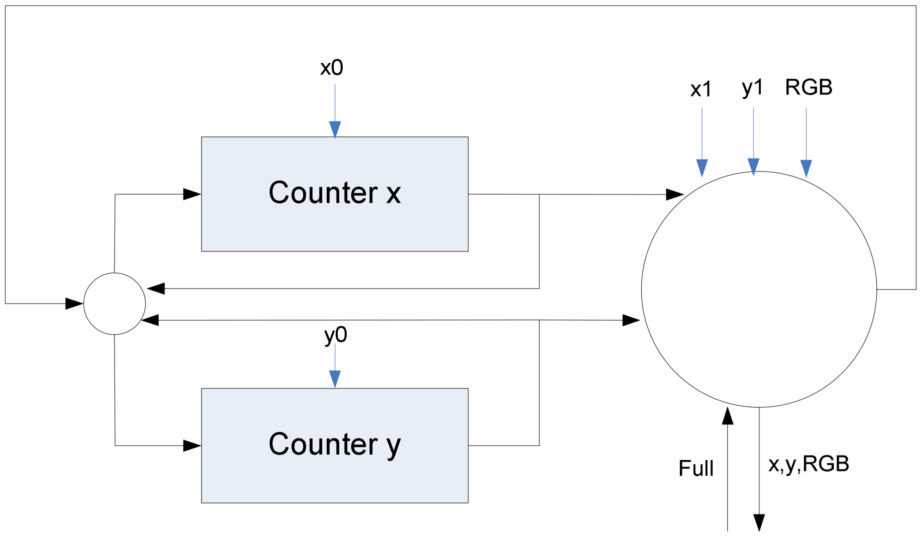


Figure 5: Blit diagram

Line Drawing

File: Line.v

Description: The line drawing module is implemented based on the well-known Bresenham algorithm. This algorithm is as follows:

```
function line(x0, x1, y0, y1)
        boolean steep := abs(y1 - y0) > abs(x1 - x0)
        if steep then
                swap(x0, y0)
                swap(x1, y1)
        if x0 > x1 then
                swap(x0, x1)
                swap(y0, y1)
        int deltax := x1 - x0
        int deltay := abs(y1 - y0)
        int error := 0
        int ystep
        int y := y0
        if y0 < y1 then ystep := 1 else ystep := -1
        for x from x0 to x1
                if steep then plot(y,x) else plot(x,y)
                error := error + deltay
                if 2×error = deltax then
                        y := y + ystep
                        error := error – deltax
```

This version of the algorithm was suitable since it does not use floating point numbers. 4 pipe line stages were created to perform calculations progressively. The final loop was implemented as an FSM that iterated over the x coordinates and calculated the required y coordinate to draw the line correctly. On each iteration (clock cycle), the same process as the blit module is used, i.e. if the output FIFO is not full, the pixel data is written and the counter(s) are incremented. If not, the counters do not change (stall).

In Figure 6, the high level view of the line module is shown. It is similar to the blit in that x / y counters are used, but the extra slope calculation is also considered and thus makes the next state logic more complicated.

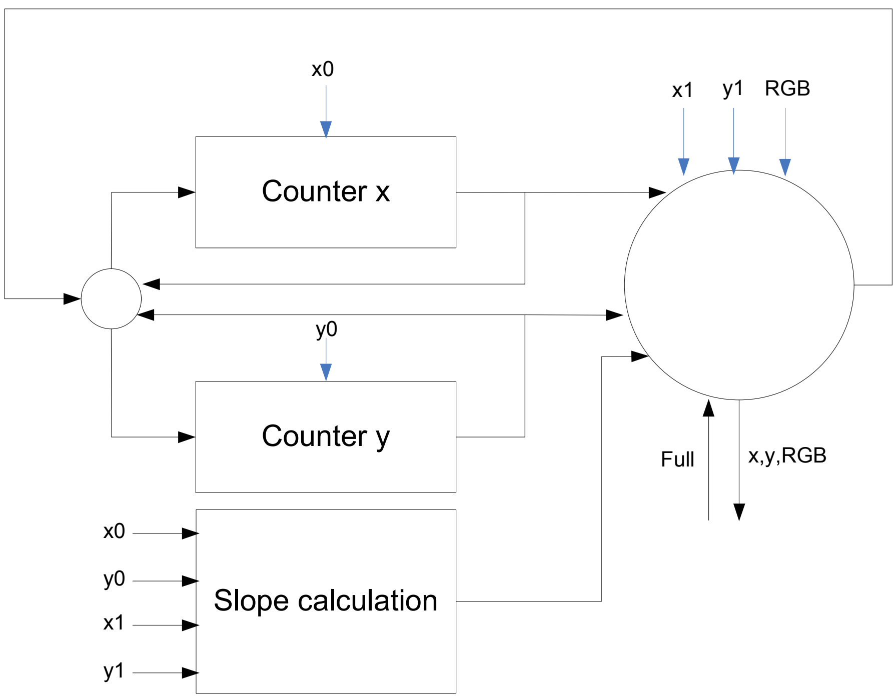


Figure 6: Line drawing

Character Drawing

File: Char.v
Description: The Character Drawing operation draws characters on the VGA display. It identifies the character to draw based on the ASCII code, and issues the address and color of the pixels to be drawn on the VGA display.

Figure 7 shows the Character operation circuit. The circuit components are:

Component	Type	Description
Countx	3-bit counter	Increments the x-coordinates of the VGA address
County	3-bits counter	Increments the y coordinates of the VGA address
X	10-bits register	store the x coordinates value
y	10-bits register	store the y coordinates value
Char_data	64-bits register	store the character data
Left Shift	64-bits left shift	Shifts the character data by 1

	register	
BRAM	Memory block	Xilinx core, acts as a ROM, stores the character data already initialized (using CoreGen)

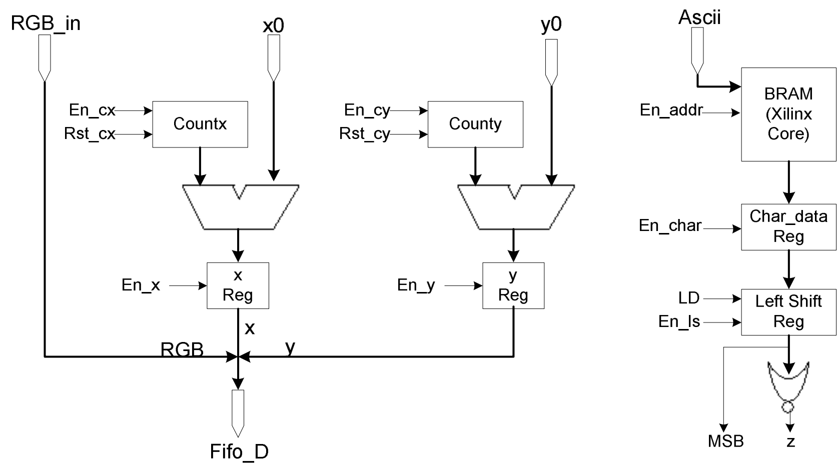
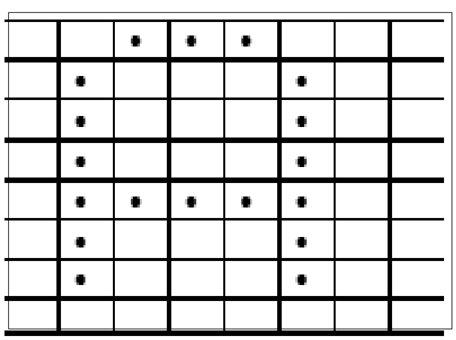


Figure 7: Character draw circuit

Each character is 8x8 pixels, therefore the character data is 64-bits, each bit representing one pixel. An active high bit indicates that the pixel is set, and on an active low bit, the pixel is not set. For an example the Character 'A' is:



0	0	**1**	**1**	**1**	0	0	0
0	**1**	0	0	0	**1**	0	0
0	**1**	0	0	0	**1**	0	0
0	**1**	0	0	0	**1**	0	0
0	**1**	**1**	**1**	**1**	**1**	0	0
0	**1**	0	0	0	**1**	0	0
0	**1**	0	0	0	**1**	0	0
0	0	0	0	0	0	0	0

Figure 8: Sample character data

The BRAM stores all uppercase and lowercase characters only. On receiving a valid data, the Character operation reads the character data and loads it into the shift register. The shift register keep shifting the data by one bit to the left, and checks the most significant bit:

- MSB=1, the pixel x, y coordinates, and RGB value are stored in the output FIFO.
- MSB=0, the pixel x, y coordinates is ignored.

The character operation keeps shifting until all the character data are zero. Figure 9 shows the data path and control path ASM chart for the character operation.

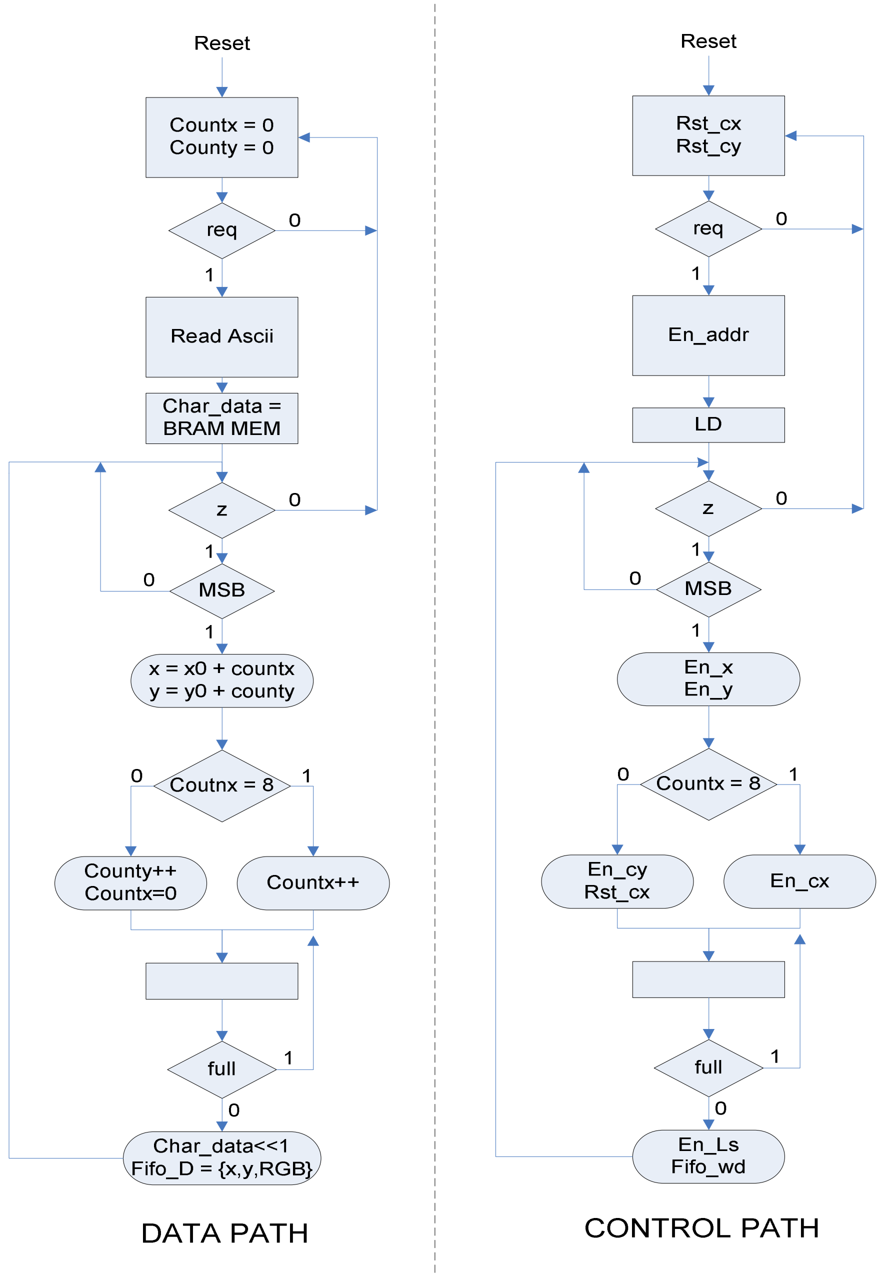


Figure 9: Character Draw Flowchart

Arbiter

File: Arbiter.v
Description: The Arbiter is the module that writes the display data from the 2D graphics modules to the frame buffer. It contains the four FIFO's, each storing data from one of the four modules. The FIFO's store the pixel information, and the arbiter performs round-robin arbitration to write pixel data on each clock cycle to the ZBT memory in the VGA display controller. The Arbiter is a Mealy state machine, implemented using Verilog. Figure 10 shows the data path ASM chart for the Arbiter. The Arbiter has 4 Mealy states,

each representing one of the operations. If the VGA memory is ready to receive (RTR) and the FIFO under consideration is not empty, the FIFO is dequeued and the data is passed to the VGA memory. The round robin technique used for the Arbiter allows higher priority to the operation of the next state over all other states, however it skips states with empty FIFO's.

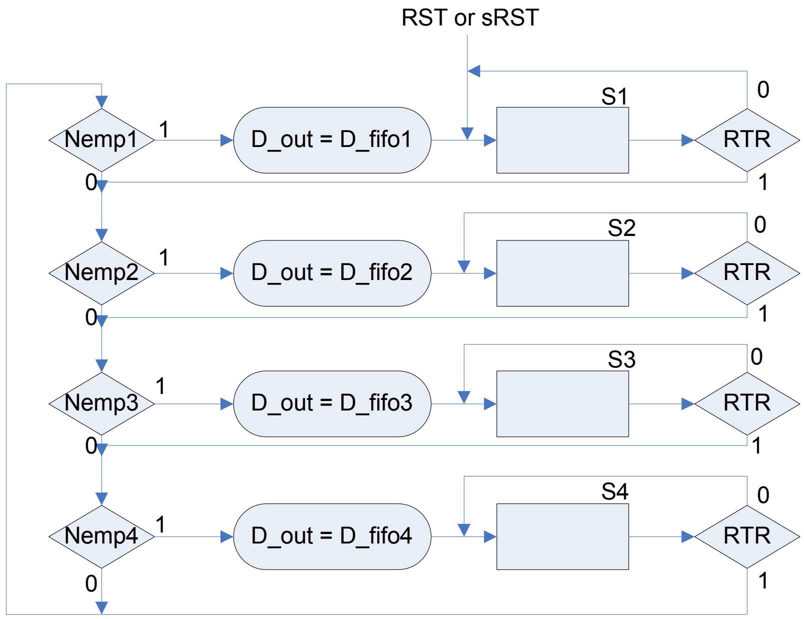


Figure 10: Arbiter Datapath

<u>FIFO</u>
File: Fifo.v (initial version), Fifo_2.v (final version)
Description: The FIFO's in this system were used within a synchronous, single clock domain. 5 FIFO's were instantiated: the input command FIFO and 4 output pixel FIFO's.

FIFO	Width (bits)	Depth	Description
1. Command FIFO	163	16	Store op code and user data
2. Line FIFO	44	16	Store pixel information from Line module
3. Blit FIFO	44	16	Store pixel information from Blit module
4. Char FIFO	44	16	Store pixel information from Char module
5. Pixel FIFO	44	16	Store pixel information from Set Pixel module

All FIFO's follow the same implementation, and are just parameterized to use different width/depth configurations. Figure 11 shows the digital circuit used to design the synchronous FIFO using Verilog (note this is not the exact synthesized circuit).

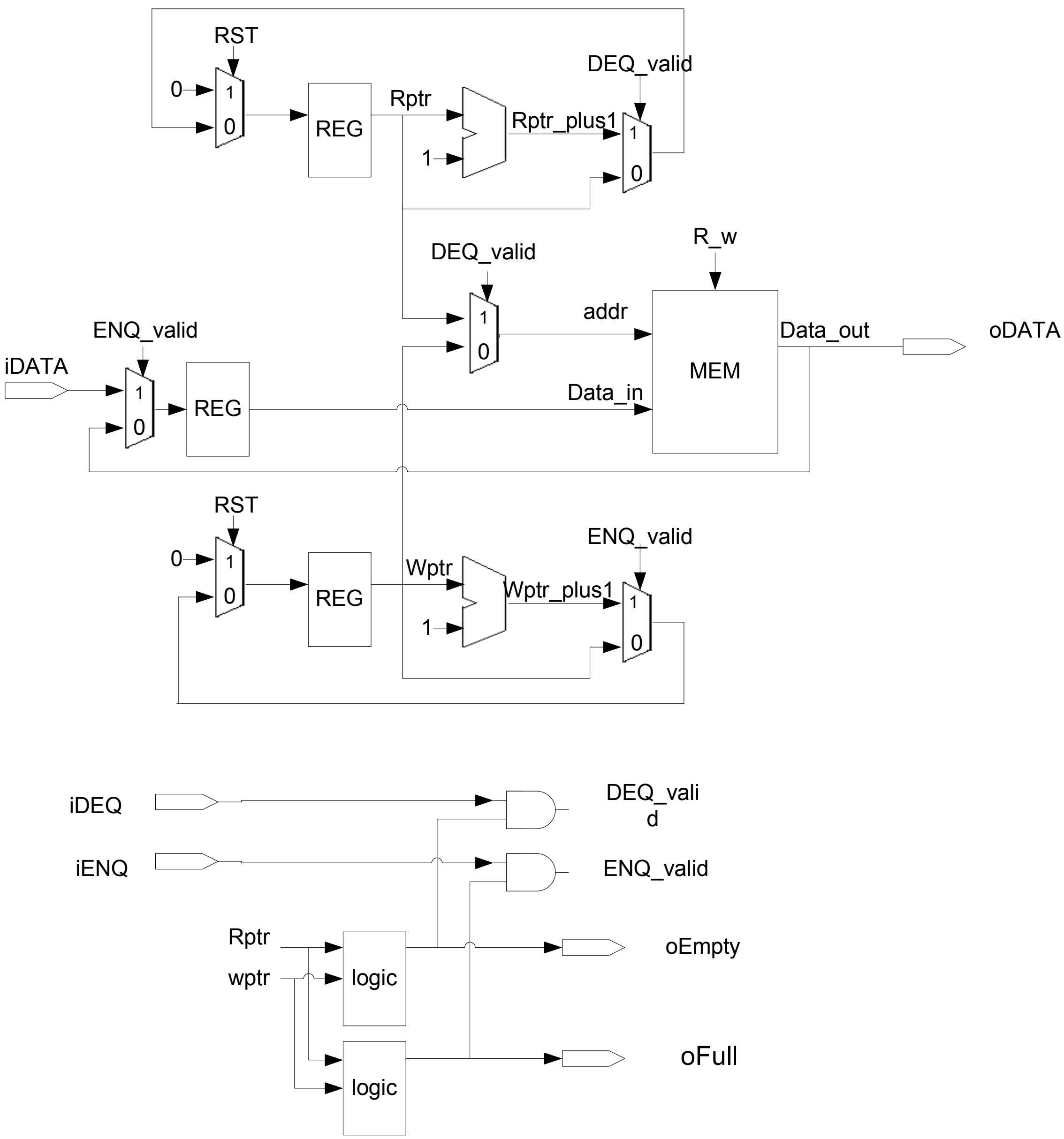


Figure 11: FIFO architecture

3.1.7.4 Software API

The following is a more technical explanation of each of the low-level software API functions.

Table 6: Software API - Low-level Functions

Function Prototype	Description
int gfx2dReady();	Returns 1 if the decoder command FIFO is not full, otherwise returns 0 (not ready).
int setData (Xuint32 x0, Xuint32 y0, Xuint32 x1, Xuint32 y1, Xuint32 RGB);	Performs register writes to set the input_data registers in the OpbInterface

	module.
int lineOp (Xuint32 x0, Xuint32 y0, Xuint32 x1, Xuint32 y1, Xuint32 RGB);	Calls setData to set line parameters, and then requests line operation via register write.
int blitOp (Xuint32 x0, Xuint32 y0, Xuint32 x1, Xuint32 y1, Xuint32 RGB);	Calls setData to set blit parameters, and then requests line operation via register write.
int charOp (Xuint32 x0, Xuint32 y0, Xuint32 ascii, Xuint32 RGB);	Calls setData to set character parameters, and then requests line operation via register write.
int pixelOp (Xuint32 x, Xuint32 y, Xuint32 RGB);	Calls setData to set pixel parameters, and then requests line operation via register write.
int gfxRegRead ();	Reads gfx2d registers and prints them to stdout.

The following describes the high level functions currently implemented in the software API.

Table 7: Software API - High-level functions

Function Prototype	Description
int clearScreen (Xuint32 RGB);	Fill the entire screen with a single colour (RGB)
int drawRect (Xuint32 x0, Xuint32 y0, Xuint32 width, Xuint32 height, Xuint32 RGB);	Draw a rectangle (unfilled) at x0,y0 with given width, height, RGB.
int fillRect (Xuint32 x0, Xuint32 y0, Xuint32 width, Xuint32 height, Xuint32 RGB);	Draw and fill a rectangle at x0,y0 with given width, height, RGB.
int drawSquare (Xuint32 x0, Xuint32 y0, Xuint32 length, Xuint32 RGB);	Draw a square (unfilled) at x0,y0 with given width, RGB.
int fillSquare (Xuint32 x0, Xuint32 y0, Xuint32 length, Xuint32 RGB);	Draw and fill a square at x0,y0 with given width, RGB.
int drawTriangle (Xuint32 x0, Xuint32 y0, Xuint32 x1, Xuint32 y1, Xuint32 x2, Xuint32 y2, Xuint32 RGB);	Draw triangle between the 3 points.
int drawStar (Xuint32 x, Xuint32 y, Xuint32 r1, Xuint32 r2, Xuint32 RGB);	Draw a star centered at x,y with inner radius r1, outer radius r2.
int rotateSquare90 (Xuint32 x, Xuint32 y, Xuint32 r, Xuint32 RGB, int clear);	Draw square at x,y with width r and rotate 90 degrees. Set clear = 1 to remove square from previous frame.
int rotateStar90 (Xuint32 x, Xuint32 y, Xuint32 r1, Xuint32 r2, Xuint32 RGB, int clear);	Draw star at x,y with radii r1, r2 and rotate 90 degrees. Set clear = 1 to remove star from previous frame.
void drawString (Xuint32 x, Xuint32 y, Xuint32 RGB, char* msg);	Draw characters in a given string starting at x,y.
void ppmOp (Xuint32 x, Xuint32 y, Xuint8 * ppm);	Blit raw ppm data to the screen starting at x,y.

3.1.7.5 Software-based Bitmap Mode

The ppmOp function takes in three parameters:

Type	Name	Description
uint32	x	The x coordinate of the top-left pixel on screen
uint32	y	The y coordinate of the top-left pixel on screen
uint8 *	PPM	The array holding the PPM data

The ppmOp function first parses the PPM header to find the width and height of the picture. The format is as follows: (MAGIC)(SPACE)(WIDTH)(SPACE)(HEIGHT)(SPACE)(MAX)(NEWLINE)(DATA).

Symbol	Size (bytes)	Description
SPACE	1	"0x20"; a space used as a header parser
MAGIC	2	The magic number set to "P6" for binary data (it can also be "P3" for ASCII data, but we do not support it)
WIDTH	1-4	The width of the picture in ASCII
HEIGHT	1-4	The height of the picture in ASCII
MAX	1-3	The max value a channel can take in ASCII. Usually this value is 255
NEWLINE	1	"0x0A"; used to represent the end of the header
DATA	WIDTH*HEIGHT*3	The raw RGB data, one bytes for each channel

Once the width and height of the picture is known, two nested for loops are created to loop through the width and height. A pixelOp is called within these loops to draw a single pixel, which is read from the RGB data in the PPM array (following the header). As the RGB values are read from the array and passed to the pixelOp, their bit endianness is reversed. This is an important step as the endianness is different in the frame buffer than it is in the PPM file.

3.2 Components on Multimedia board

The on-board components used were:

- ZBT RAM modules (2 instances as shown in Figure 1). Source documentation: http://www.eecg.toronto.edu/~pc/courses/edk/doc/ZBT_k7n163601a.pdf
- VGA DAC: Documentation at http://www.xilinx.com/bvdocs/userguides/ug020.pdf

3.3 External devices

The only external device to the Multimedia board was a Samsung SyncMaster 710N 17" LCD monitor.

4. Description of Design Tree

4.1 Directory Structure and Files

/ (Root directory)	
clear_screen.tcl	TCL script to fill screen with a single colour (run in XMD)
get_dbg.tcl	TCL script to dequeue (and read) debug data from internal debug FIFO (run in XMD)
get_regs.tcl	TCL script to read all gfx2d registers (run in XMD)
load.tcl	TCL script to download executable.elf to board (run in XMD)
test_blit.tcl	TCL script to test blit operation (run in XMD)
test_char.tcl	TCL script to test char operation (run in XMD)
test_line.tcl	TCL script to test line operation (run in XMD)
test_pixel.tcl	TCL script to test pixel operation (run in XMD)
test_gfx2d_linker_script.ld	Links executable file into ZBT memory instead of Microblaze BRAM cache.
system.mhs	Hardware specification of system.
system.mss	Software specification of system.
system.xmp	XPS project file.

/code (Application software)	
gfx2d.h	API for 2D Graphics Engine (C header file) – include in any application that uses the graphics engine.
pic.h	Sample PPM data stored as char[] variable.
test_gfx2d.c	C code containing test and demo code for graphics engine.

/data	
system.ucf	Contains pin-out information for use with Multimedia board.

/pcores/gfx2d_v1_00_a/data/ (Data files for gfx2d pcore)	
gfx2d_v2_1_0.bbd	Black-box descriptor file for graphics 2D pcore – required to instantiate BROM for character draw.
gfx2d_v2_1_0.mpd	Descriptor file for pcore – defines interface to system
gfx2d_v2_1_0.pao	Port analyze order file – defines which source files are compiled and in which order

/pcores/gfx2d_v1_00_a/hdl/verilog (Source code for gfx2d pcore)	
Arbiter.v	Arbiter module
Blit.v	Blit module
brom.v	Wrapper for brom netlist containing character data.

Char.v	Char module
Decoder.v	Decoder module
Fifo_2.v	Fifo module used for all Fifo's
gfx2d.v	Top-level gfx2d module
Line.v	Line module
OpbInterface.v	OPB bus interface module

/pcores/gfx2d_v1_00_a/netlist/ (Netlist files for gfx2d pcore)	
brom.edn	Synthesized brom that contains character data as used in Char module

/pcores/gfx2d_v1_00_a/sim/ (Simulation files for gfx2d modules)	
blit/	Testbench and Modelsim do files to test Blit module
gfx2d/	Testbench and Modelsim do files to test gfx2d module
line/	Testbench and Modelsim do files to test Line module

/pcores/display_v1_00_a/sim/ (Bit Mapped Mode SVGA pcore)	
data/	Pcore data files
hdl/verilog/	Verilog source code for pcore

4.2 Instructions to synthesize, download and run 2D graphics system

1) Connect Multimedia board to PC via parallel cable, etc. (follow instruction in tutorial module m01)
   - We used the Multimedia board based on the Virtex-II XC2V2000-FF896 FPGA
2) Connect monitor (we used a 17" LCD monitor) to Multimedia SVGA port
3) Open system.xmp in XPS (we used version 8.2.02i)
4) If choosing resolution other than 640x480 (UNTESTED!):
   a. Modify #define's in code/gfx2d.h
   b. Modify `define RESOLUTION_H and `define RESOLUTION_V in pcores/gfx2d_v1_00_a/hdl/verilog/gfx2d.v
   c. Modify `define RESOLUTION_H and `define RESOLUTION_V in pcores/display_v1_00_a/hdl/verilog/SVGA_TIMING_GENERATION.v
5) If modified anything in pcores/: select Hardware->Clean Hardware
6) Select Hardware->Generate Bitstream (ensure board is powered on first)
7) Once bitstream successfully downloads, select Debug->Launch XMD
   - Ensure reset switch on board is OFF (low)
8) Type "source load.tcl" to load program code into memory via JTAG.
   - Ensure the memory region used to store the program is above 0x20200000
   - If not, you will need to regenerate the linker script:
     i. Close XMD
     ii. Select the Applications tab on the left hand menu of XPS

iii. Right click on "Project: test_gfx2d"
iv. Select "Generate Linker Script"
v. Ensure all memory regions are set to ZBT_512Kx32_*
vi. Set the stack size to 0x81000
vii. Click "Generate"
viii. Repeat from Step 6

9) Type "run" to run program code.

5. References

1) Bit Mapped Mode SVGA example source file
http://www.xilinx.com/products/boards/multimedia/docs/examples/BM_MODE_SVGA.zip .

2) Multimedia Board user guide
http://www.eecg.toronto.edu/~pc/courses/432/2004/handouts/Multimedia_UserGuide.pdf

3) Multimedia Board datasheet
http://www.eecg.toronto.edu/~pc/courses/432/2004/handouts/Multimedia_Schematics.pdf

4) ZBT RAM memory controller
http://www.xilinx.com/products/boards/multimedia/docs/examples/ZBT.zip

5) ZBT RAM behavioral simulation model
http://www.eecg.toronto.edu/~pc/courses/edk/modules/6.3/m08.zip

6) ZBT RAM data sheet
http://www.eecg.toronto.edu/~pc/courses/edk/doc/ZBT_k7n163601a.pdf