Interactive Graphical Processing Platform with Application Support

2D GPU Platform with Hardware-Accelerated Features

Stephen Just - sajust@ualberta.ca
Stefan Martynkiw - smartynk@ualberta.ca
Mason Strong - mstrong@ualberta.ca

Designing a platform to easily add graphics support to interactive applications.

Abstract

The aim of the project was to produce a simple graphics platform to allow the creation and manipulation of graphics on a 640x480 display with 256 colors on the DE2.

The hardware is maximally capable of rendering frames at up to 60 frames per second. Our hardware makes use of multiple memory devices to maximize the available memory bandwidth for application code and image drawing operations. Tearing on the video output is eliminated by an arrangement of bursted memory transfers combined with a FIFO queue to control pixel transmission. 16-bit colour is achieved via a colour palette system allowing any 256 colours to be visible at one time.

We implemented simple drawing operations in hardware to draw shapes such as rectangles, circles, and lines. These operations were implemented as NIOS/II custom CPU instructions. Implementing these instructions in hardware greatly simplified the generation of graphics for the programmer while enabling a performance boost on an order of magnitude.

We used hardware to copy graphics from external SD Card storage to a frame buffer, allowing the use of bitmaps.

Human-computer interaction is facilitated with two Sega Genesis 3-button controllers. A custom PCB and VHDL logic was developed to interface these controllers to the GPIO headers on the DE2.

All of this functionality is wrapped in an easy-to-use C API. To demonstrate the capabilities of our system and the graphics API, we built a version of Pong which uses bitmaps, layering, and the Sega Genesis controller inputs.

Table of Contents

Functional Requirements

This project was to create a platform with which simple applications could be built to make use of a graphical display.

At a high level, the graphics engine on this system supports one or more pixel buffers in memory which can be composited into a video frame. It will output video at 640x480 resolution using 256 configurable on-screen colors. The drawing layers support easy-to-use graphical primitive instructions. When compositing layers, one colour can be designated as a “transparent” colour, enabling layers to be drawn on top of each other.

The graphics engine supports a number of simple operations in hardware:

- Drawing shapes (line, rectangle, circle, pixel)
- Copying a window from a pixel buffer into another pixel buffer, with or without transparency
- Colour palette decoding during video output
- VSync synchronized frame swapping

Other more complex operations are supported by combining the above in software:

- Drawing text
- Drawing shapes (triangles, filled circles, rounded rectangles)

Using the above combination of operations, it is possible to represent a large number of graphical elements at a framerate of at least 20 frames per second, and potentially up to 60 frames per second depending on the software optimizations used and the number of shapes being drawn.

In order to demonstrate the above, we implemented a simple application, the game of Pong. This made use of many of the capabilities above, through some graphical effects designed to show off the platform’s capabilities rather than to add to gameplay. This Pong game runs on an instance of a Nios II running in the FPGA. To conserve memory and CPU cycles, this Nios II does not run MicroC-OS/II, but instead simply uses Altera’s HAL library with bare-metal C, running the pong game in a tight loop.

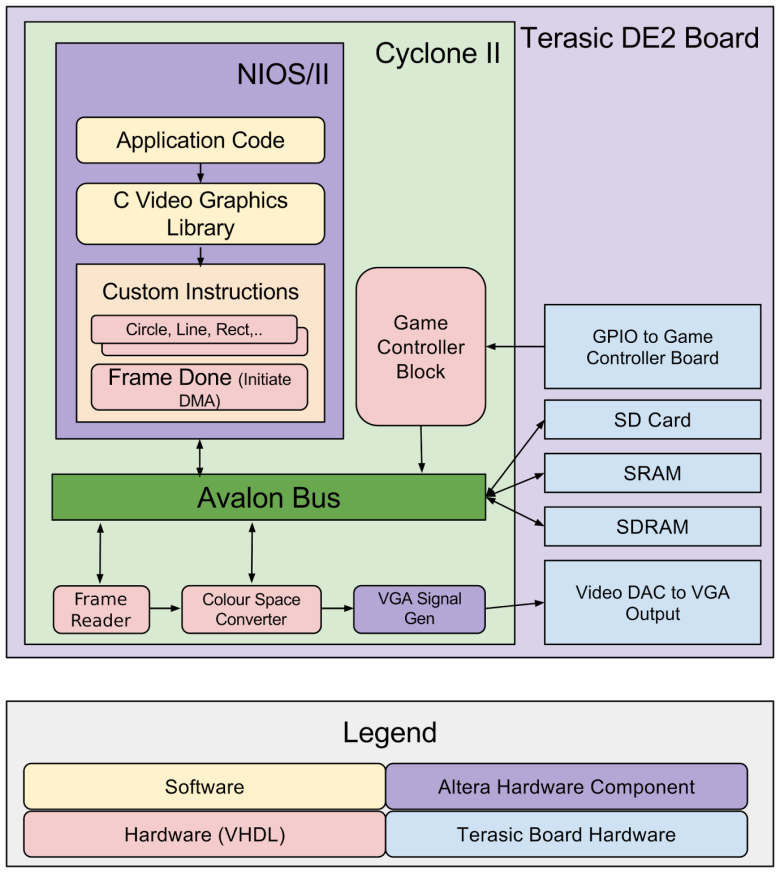
To handle input for this application, we implemented an interface to connect to controllers from the Sega Genesis game console. This module reads the controller inputs once per video frame, and make the input data available to the running application as a memory-mapped device. We chose the Genesis controller due to its low cost, simplicity, and documentation. The Sega Genesis controller simply contains a 4x2-input multiplexer chip and some pull-up circuitry on each button. This approach is much simpler than attempting to interface with a device using a more modern protocol like RS-232, PS/2 or USB.

We met most of the requirements that we had set out for ourselves in the earlier design phases of the project. A summary of these requirements exists in the following table.

Feature	Achieved?	Comments
Signalling the completion of a frame	Yes	
Loading a colour palette into memory	Yes, with design change	The design changed from having the colour palette stored in the upper ⅜ of SRAM to being stored in instantiated Altera Dual-Port RAM in the colour palette converter block.
Applying an alpha mask to a graphics layer	Yes	The alpha mask is applied during layer compositing/copy from the sublayer to the finalized output layer.
Drawing shapes (line, rectangle, pixel) to a graphics layer	Yes	Filled triangles and circles are accomplished by having software call multiple line-draws.
Drawing a texture to a graphics layer	Yes	The system supports drawing bitmaps to arbitrary framebuffers and offsets to the framebuffer. Bitmaps can also have a transparent colour set during the DMA to enable transparent regions.
Drawing a texture to the background layer with an offset	Yes	
Loading a texture into memory	Yes	
Loading a sprite into memory	No, but can be simulated	The original design has support for hardware sprites. The idea was that small-sized layers would be DMA'd to a framebuffer, and hardware would keep track of where the sprites were. This can be simulated with the as-built hardware by making one layer per sprite, and keeping track of the coordinates in software.
Drawing a sprite to a position and layer		
Setting the background velocity in pixels per frame	No	This feature was intended for scrolling backgrounds. The original design had provisions for scrolling backgrounds.

Design & Description of Operation

To describe the design and operation of our video platform, let us first begin with a diagram describing the organization of the hardware components.



The diagram shows the overall organization of the system. The NIOS II processor runs the application code that calls the C video graphics library. The graphics library invokes our graphical hardware primitives. Each hardware primitive is implemented as a custom CPU instruction.

The custom instructions block the CPU for a variable length of CPU cycles dependent on the number of pixels plotted.

Each CPU instruction component also contains several registers. These registers are organized as follows, typically.

- Pixel buffer base address
- Graphical Parameters (bounding coordinates for a rectangle, center/radius for a circle, etc..)
- Fill colour

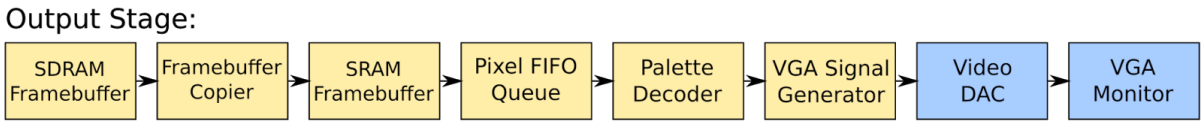
Once the CPU custom instruction is called, a state machine described as {IDLE, RUNNING} begins. When the instruction is in IDLE state, the registers within the instruction hardware are being populated. When the instruction is RUNNING, the hardware is making Altera Memory-Mapped Master writes into a pixel framebuffer. The particular framebuffer chosen for writing is controlled by the user according to the scheme described below:

There are several framebuffers that can be written to. One framebuffer exists on the SRAM. This framebuffer is continuously read from by the VGA pixel drawing hardware (consisting of the frame reader, colour space converter, and VGA Signal Generator blocks in the block diagram above). Writing directly to the SRAM during a frame draw will cause noticeable visible artifacting and tearing.

For this reason, our system design has the user writing graphical primitives to a framebuffer in the SDRAM. The SDRAM is organized into a number of 640x480 framebuffers. The base addresses of these framebuffers are passed into the registers in the custom hardware.

Once the user is done writing into the SDRAM framebuffer and wishes to display the picture onto the screen, the user needs to call the ALT_CI_CI_FRAME_DONE instruction. This instruction performs a DMA memory copy from the primary SDRAM framebuffer into the SRAM starting during the vertical blanking interval.

A more technical version of the VGA output pipeline is shown below.



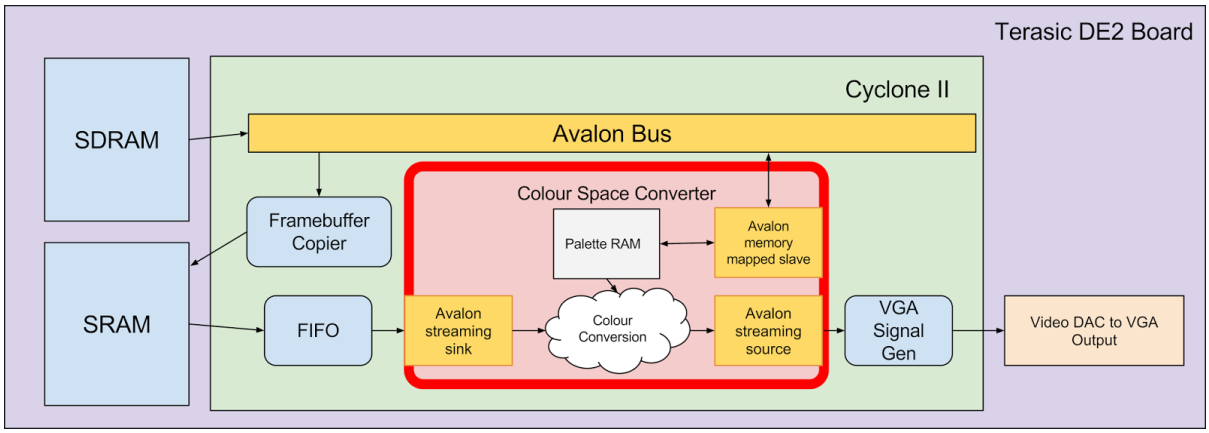
In detail, the video output pipeline works as follows. The 512 kB SRAM is capable of holding one frame of video at 8-bit colour depth as well as other data in the remaining space. The framebuffer section of the SRAM contains 640x480 8-bit words, with each word containing a palette colour number. A component reads the data out of the SRAM in bursts into a FIFO. This FIFO is emptied at a slower rate to feed the video output pipeline at the same rate as the pixel clock, 25.2 MHz. The flow of pixel data beyond this point all follows the Avalon-ST interface specification. The data from the FIFO is fed into the Palette Decoder component, which takes the incoming 8-bit data and converts it to 16-bit colour data using an on-chip 512 byte memory block. The remaining components in the video output pipeline convert the 16-bit colour data to the 30-bit format required by the VGA Signal Generator component, which generates the necessary signal timing to drive a display at 640x480 @ 60 Hz. These final components are provided by Altera.

The Framebuffer Copier component copies from the SDRAM to SRAM. This transfer is triggered by a Nios II custom instruction that activates this component. Once it has been triggered, this component immediately reads a burst of pixels from the SDRAM buffer, then waits for the end of a frame to be read from SRAM into the pixel FIFO, and then immediately starts writing that first burst of pixels to the SRAM. While the data is being written out to the SRAM, another burst of pixel data is read from the SDRAM in parallel. This continues in a

similar fashion until the entire SDRAM buffer has been copied to SRAM. Writes to SRAM are interrupted when the output FIFO gets low on data, at which point some pixels will be read from SRAM. Because the system clock orchestrating these memory transfers from SDRAM to SRAM is significantly faster than the pixel output clock, it is possible to update all of the data in SRAM before the pixel output pipeline needs to read it, thus eliminating tearing on the video output.

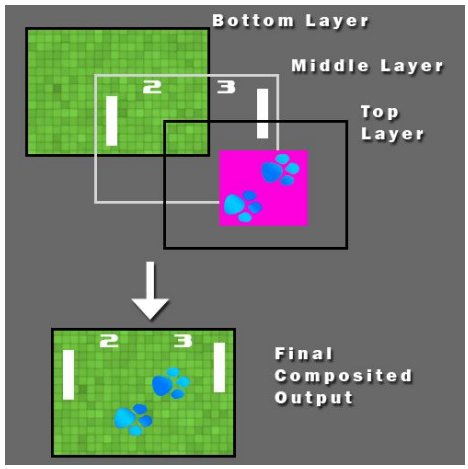
In more detail, the Palette Decoder works as follows: each 8-bit word given from the pixel FIFO is mapped to a 16-bit RGB value (in the RGB 565 format). This allows, for example, a frame to show 256 shades of blue, if required. The default active colour palette defaults to one that provides a standard 8-bit to 16-bit colour mapping. Changing the active palette will not be synchronized with video frame updates, so application developers must take care to provide a black frame, or some other data that will appear acceptable even with an incompletely updated colour palette.

A diagram of the colour palette shifter hardware is shown below.



As mentioned earlier, there can be several framebuffers stored in the SDRAM. Our project also supports layering (with transparency). In the SDRAM there is a primary framebuffer which is copied to the output stage. Elsewhere in the SDRAM are other framebuffers. Our graphics API exposes a custom instruction to copy buffers in a user-friendly manner. This instruction performs a hardware DMA between two SDRAM framebuffers. This instruction also takes a parameter which allows a particular colour to be set as the “transparent” colour.

This feature enables graphical compositions such as the one depicted below.



Each of these hardware features involve a storage cost in the SDRAM. The overall function of the SDRAM in the system is to provide a scratch space for compositing frames. These frames, when composited, are sent to the SRAM when timing allows, as previously described.

The C graphics API is quite easy to use. An example usage is shown below, followed by a table of features.

```
#include <io.h>
#include <system.h>
#include <sys/alt_stdio.h>
#include "sys/alt_timestamp.h"
#include <string.h>

#include "graphics_commands.h"
#include "palettes.h"

int main()
{
    graphics_init();
    graphics_clear_screen();
    switch_palette(&palette_ega);
    /* Red Rectangle */
    graphics_draw_rectangle(graphics_get_final_buffer(), 0, 0, 640, 480, 15);
    ALT_CI_CI_FRAME_DONE_0;


    /* Filled Circle in Hardware, negative fills */
    graphics_draw_circle(graphics_get_final_buffer(), 640/2, 480/2, 239, 3, 0);
    /* Filled Circle in Software */
    graphics_draw_circle(graphics_get_final_buffer(), 640/2, 480/2, 239, 4, 1);
    /* Line diagonally across */
    graphics_draw_line(graphics_get_final_buffer(), 0, 0, 640, 480, 5);

    ALT_CI_CI_FRAME_DONE_0;

    /* Fonts. */
    print2screen(graphics_get_final_buffer(), 20, 20, 6, 2, "Hello, World!");

    graphics_draw_triangle(graphics_get_final_buffer(), 15, 112, 300, 112, 170, 240, 1,
9);
    ALT_CI_CI_FRAME_DONE_0;
    return 0;
}
```

The C-API provides the following functions.

```
char graphics_init();

pixbuf_t *graphics_get_final_buffer();

void graphics_draw_pixel(pixbuf_t *pixbuf, int x, int y, unsigned char color);

void graphics_draw_rectangle(pixbuf_t *pixbuf, int x1, int y1, int x2, int y2, unsigned
char color);

void graphics_draw_line(pixbuf_t *pixbuf, int x1, int y1, int x2, int y2, unsigned char
color);

void graphics_draw_circle(pixbuf_t *pixbuf, int cx, int cy, int radius, int color, int
filled);

void graphics_draw_rounded_rect(pixbuf_t *pixbuf, int x1, int y1, int x2, int y2, int
radius, int filled, unsigned char color);

void graphics_clear_screen();

void graphics_clear_buffer(pixbuf_t *pixbuf);

void graphics_draw_triangle(pixbuf_t *pixbuf, int x1, int y1, int x2, int y2, int x3,
int y3, int filled, int color);


/* Fonts */
extern char font8x8_block[][8];
extern char font8x8_basic[][8];

void draw_letter(pixbuf_t *pixbuf, int y1, int x1, int color, int pixel_size, char*
letter);
void print2screen(pixbuf_t *pixbuf, int x1, int y1, int color, int pixel_size, char*
string);

void copy_buffer_area(pixbuf_t *source, pixbuf_t *dest, rect_t *source_area, point_t
*dest_offset);
void copy_buffer_area_transparent(pixbuf_t *source, pixbuf_t *dest, rect_t *source_area,
point_t *dest_offset, unsigned char t_color);
```

Overall, the C-API is very easy to use and powerful. Further discussion on the design and implementation of the C-API is located in the ***Software Design*** section of this report.

Bill of Materials

The physical parts needed for our project surround the adapter board we will be making to adapt the SEGA Genesis controller to the GPIO header on the DE2, as well as external components required to connect to a VGA monitor.

Part	Qty	Cost/Unit	Datasheet	Description	Supplier / P/N
DB9 Male Connector	2	$2.59	http://www.norcomp.net/rohspdfs/Connectors/18Y/182/182-yyy-113Ryy1.pdf	DB9 Male, Through hole, Right angle	DigiKey / 182-09ME-ND
40 Pin shrouded header	1	$1.38	http://sullinscorp.com/catalogs/145_PAGE118_.100_SBH11_SERIES_MALE_BOX_HDR_ST_RA_SMT.pdf	2.54mm pitch male header with keyed shroud	DigiKey / S9175-ND
40 Pin Ribbon Cable	1	$2.63	http://www.assmann.us/specs/AWG28-40_G_300.pdf	Ribbon cable with 2.54mm pitch 40 pin connectors at either end	DigiKey / H3CCS-4006G-ND
Circuit Board	1	$11.25	https://oshpark.com/shared_projects/PhQrCqc9	PCB, Plated Vias, Solder mask, Silkscreen	OSHPark
VGA Cable	1	$9.25	http://www.assmann.us/specs/AK532-2-R.pdf	VGA Cable, 2m	DigiKey / AE10183-ND
Sega Genesis Controller	2	$30.00	N/A	3-button Controller	Amazon
Altera / Terasic DE2 Development Board	1	$517.72	http://www.terasic.com.tw/cgi-bin/page/archive.pl?No=30	Cyclone II FPGA Development Board	Terasic

Total Cost: $607.41

Note: 5V 2A power supply and video monitor required, not included in total

Available Source

During our evaluation of re-usable design units, we found that, in most cases, that it is typical for the VGA output block to be documented. This block, shown in [1], is typical to most output blocks: it simply takes a framebuffer (assuming proper memory bandwidth considerations) and generates the necessary analogue colour burst, Hsync, and Vsync signals.

We have found that the Terasic DE2 as well as the development environment given in the lab provides us with a pre-existing analogue output block [2] that generates the necessary signals to drive a monitor. Our search of opencores.org, as well as other sources on the internet led us to VGA output blocks such as [1] and [7]. Each of these blocks support configuration options that are unnecessary for our platform. As well, the higher resolution modes on each of [1] and [7] are easily capable of placing memory bandwidth demands that our development board cannot meet.

Using the provided VGA generation block allows us to focus on the programmer-facing aspects of the graphics generation. Because we are building a hardware graphics platform with a custom API, we do not have to design the system to interface with any other existing software API, such as a Linux framebuffer driver, for example. This enables us to pick and choose features as desired. Our main feature-set inspiration is [2], the Gameduino, another FPGA-based video generator.

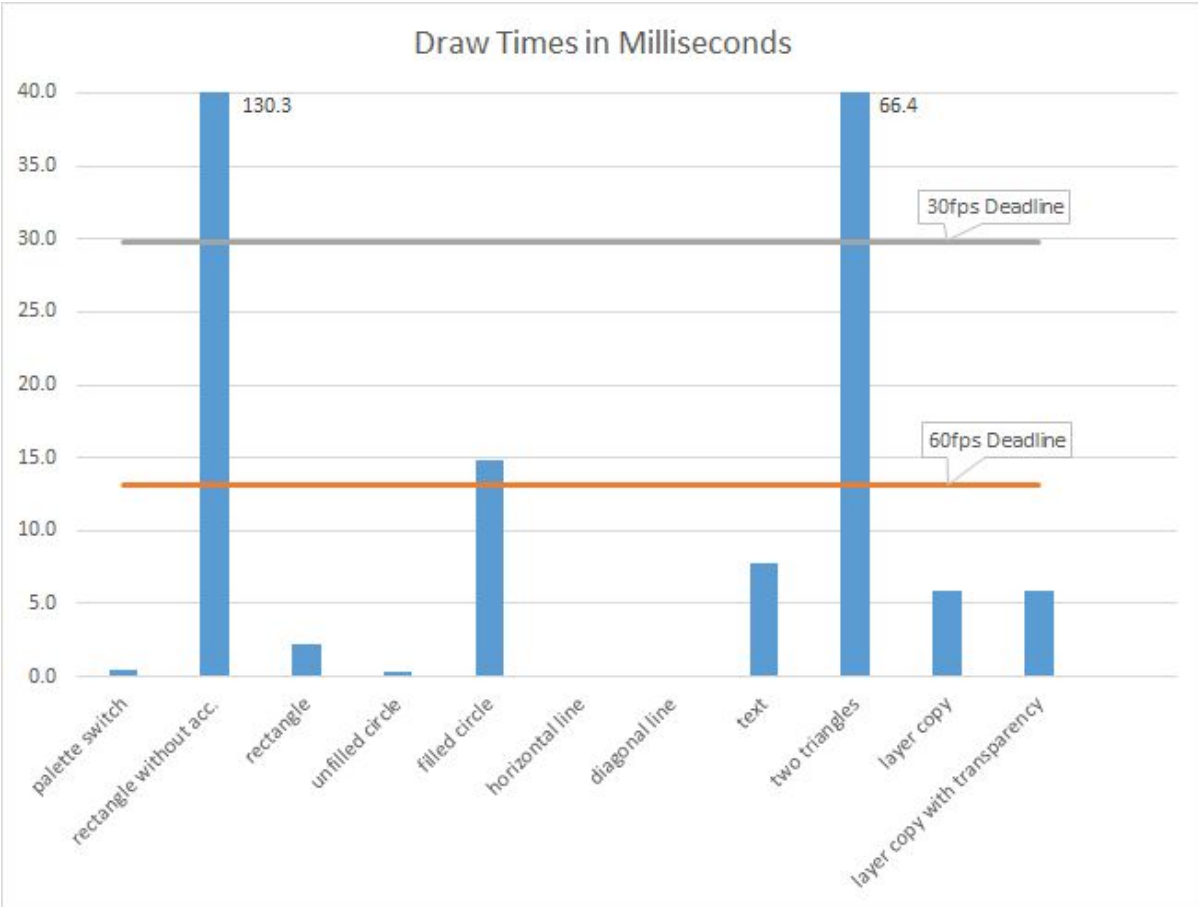
The gameduino is designed to be used by an Arduino over an SPI bus, providing high level drawing, sprite, and collision-detection primitives. As well, the Gameduino provides sound generation capabilities. The gameduino is designed to be synthesized on a Xilinx Spartan 3-calibre board with minimal supporting hardware. The Gameduino supports 400x300 resolution in 512 colours, with a 512x512 character background, and is exposed as a 32KiB memory to the SPI bus.

Datasheet

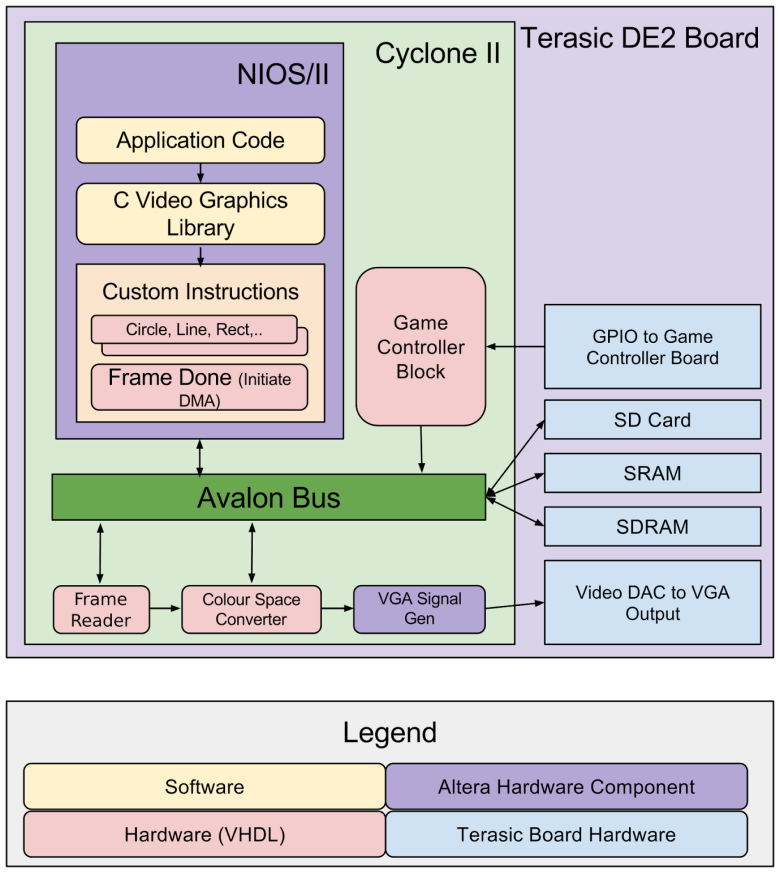
Performance

On this system, the most important measurement of performance is how quickly graphical operations can be performed. The number of operations that can be performed while maintaining a particular frame rate can be estimated by determining the worst-case execution time of an operation. This worst-case execution time relates to how many pixels the system attempts to draw. For most operations, this worst-case time occurs when the operation attempts to redraw an entire frame (the exception being lines, which are unable to redraw a complete frame at once).

The chart below shows the time in milliseconds taken to complete various worst-case drawing operations. The 60fps and 30fps deadlines represent the time allowed for a combination of commands to still maintain the specified frame-rate.



Block Diagram



Operating Conditions

Parameter	**Value (Typical)**	**Units**
Supply Voltage	9	V
Supply Current	470	mA
Operating Temperature	0 - 85	°C

IO Signals

With the exception of the external interface to connect to game controllers and to a VGA monitor, all peripherals are on the DE2 board [6].

Signal	Direction	Description	Location
CLOCK_50	input	System clock input	FPGA to Board
CLOCK_27	input	Video clock input	FPGA to Board
DRAM_ADDR[11..0]	output		FPGA to Board
DRAM_BA_0	output		FPGA to Board
DRAM_BA_1	output		FPGA to Board
DRAM_CAS_N	output		FPGA to Board
DRAM_CKE	output		FPGA to Board
DRAM_CLK	output		FPGA to Board
DRAM_CS_N	output		FPGA to Board
DRAM_DQ[15..0]	in/out		FPGA to Board
DRAM_LDQM	output		FPGA to Board
DRAM_UDQM	output		FPGA to Board
DRAM_RAS_N	output		FPGA to Board
DRAM_WE_N	output		FPGA to Board
SRAM_ADDR[17..0]	output		FPGA to Board
SRAM_DQ[15..0]	in/out		FPGA to Board
SRAM_WE_N	output		FPGA to Board
SRAM_OE_N	output		FPGA to Board
SRAM_UB_N	output		FPGA to Board
SRAM_LB_N	output		FPGA to Board
SRAM_CE_N	output		FPGA to Board
FL_ADDR[21..0]	output		FPGA to Board
FL_CE_N	output		FPGA to Board
FL_OE_N	output		FPGA to Board

FL_DQ[7..0]	in/out		FPGA to Board
FL_RST_N	output		FPGA to Board
FL_WE_N	output		FPGA to Board
VGA_R[9..0]	output		FPGA to Board
VGA_G[9..0]	output		FPGA to Board
VGA_B[9..0]	output		FPGA to Board
VGA_R_EXT	output	VGA Connector Pin 1	Off-board
VGA_G_EXT	output	VGA Connector Pin 2	Off-board
VGA_B_EXT	output	VGA Connector Pin 3	Off-board
VGA_CLK	output		FPGA to Board
VGA_BLANK	output		FPGA to Board
VGA_HS	output	VGA Connector Pin 13	Off-board
VGA_VS	output	VGA Connector Pin 14	Off-board
VGA_SYNC	output		FPGA to Board
SW[17..0]	input		FPGA to Board
KEY[3..0]	input		FPGA to Board
GPIO[35..0]	in/out	Sega Genesis Controller interface (Spec is 5V, operable at 3.3V)[5]	Off-Board

Power Consumption

This system has no idle or low power states. The power consumption in the operating mode is detailed below. When the system is powered off, it draws no power.

Parameter	**Value**	**Units**
Voltage	8.93	V
Current	450	mA
Power Used	4.02	W

VGA Signal Timing

Constant	Value
H_ACTIVE	640 (pixel clock cycles)
H_FRONT_PORCH	16
H_SYNC	96
H_BACK_PORCH	48
H_TOTAL	800
V_ACTIVE	480 (H_TOTAL * pixel clock cycles)
V_FRONT_PORCH	10
V_SYNC	2
V_BACK_PORCH	33
V_TOTAL	525

$$clock\ cycles\ per\ frame = H\ TOTAL\ \times\ V\ TOTAL = 420\ 000$$

$$\frac{pixel\ clock}{clock\ cycles\ per\ frame} = refresh\ rate$$

$$\frac{25\ 200\ 000}{420\ 000} = 60.0\ \text{Hz}$$

SRAM Memory Map

SRAM Address Range	Data
`0x00000 - 0x257FF`	Pixel buffer

Background Research

The background research for this project falls within several spheres of investigation. One investigatory sphere centers around the VGA output block which displays the contents of the SRAM framebuffer to the VGA adapter. The research surrounding the output blocks centers on the links [1]-[4] in our References section.

We found [1] to be a fully-featured VGA output system supporting multiple resolutions. We found that the larger resolutions supported by this block may be too taxing for the clock and

timing restraints imposed by our DE2 system. We have considered that, in the event that we cannot get the supplied Altera VGA output block to work correctly, we will take the VHDL provided in [1] and strip out all support for resolutions that are not 640x480.

Nevertheless, the solution provided in [1] does not provide a high-level interface to the programmer: it is still up to the programmer to manually populate the framebuffer if he wishes to implement object (sprite) movement, collision detection, and drawing primitives.

During our research, we also found the Gameduino, a “game adapter for microcontrollers” [2]. The Gameduino supports 400x300 in 512 colours, background graphics (512x512), 256 background sprites (each with 4 colour palette), and foreground sprites. The foreground sprites support 16x16 size, up to 256 colours, four-way rotate and flip, and collision detection. The Gameduino can be programmed via an SPI interface from an Arduino with minimal wiring between the two. We were most impressed with the feature set and programmer-facing API provided by the hardware. We aim to replicate many of the Gameduino’s features, but with a higher resolution.

As well, we researched the VGA timing specifications as seen in [3]-[4]. We became familiar with some of the timing constraints inherent in the VGA standard. Once we discovered that pre-made VGA blocks exist, we became concerned primarily with the timing constraints between the SRAM and the black-box VGA output block.

Another investigatory sphere of research centered around the algorithms to draw primitives such as lines and circles. Our research in [11] led to Jack Bresenham’s algorithms. We used Bresenham’s line and circle algorithm pseudocode to write VHDL that plots lines and circles in hardware.

Software Design

The software running on our system consists of the following components:

1. NIOS II software project.
    a. This is where the user will put their code
2. Video_System_BSP
    a. There is one BSP needed for every application running on our platform.
3. Video_System_Graphics_Library
    a. This is a NIOS/II library project that must be linked to item 1. This project contains the C-API.

Our Pong demo application also follows the above scheme.

C-API Design

The C-API is implemented as a NIOS/II library project. The overall layout of the API is set up so that the API's constants and functions are organized by concerns.

File	Contents
Graphics_defs.c, graphics_defs.h	Definitions of typedef'd structures for pixelbuffers (pixfbufs, aka: framebuffers), points, rectangles, and colour palettes
Graphics_commands.c, graphics_commands.h	Definition and implementations of commands to: ● Plot ○ Pixel ○ Rectangle ○ Line ○ Circle ○ Rounded Rectangle ○ Triangles ● Write a character to screen ● Write a string to screen ● DMA copy buffer areas from one pixbuf to another. ○ Without transparency ○ With Transparency
Graphics_layers.c, graphics_layers.h	Definition and implementations of commands to: ● Allocate and add a pixbuf to the rendering stack ● Get a pointer to the origin point of the pixel buffer
Palettes.c, palettes.h	Definition of colour palettes: ● EGA 16-colour palette ● RGB-332 colour palette ● Test palettes containing magenta, various shades of blue and red Commands to: ● Switch the graphical output to a specified colour palette ● Print a colour palette definition to stdout.
Sdcard_ops.c, sdcard_ops.h	Commands to: ● Load and parse a bitmap file from an sdcard into a pixel buffer ● Load a file (without regard for format) into a void* buffer.

	○ This function is used to read the palette file accompanying the bitmap. This functionality depends on libEFSL.
Flash_ops.c, flash_ops.h	Same as above, except loading from the flash memory on the DE2 board instead of the SD Card. This functionality depends on the Altera ZipFS library.

Each graphical operation operates on a ***pixbuf_t***. This data structure defines a region in memory which is a framebuffer. Each cell in the 640x480 array is one byte corresponding to the colour to be displayed at that screen location. The pixbuf_t type is defined as follows:

```
typedef struct pixbuf_t
{
    void *base_address;
    unsigned short width;
    unsigned short height;
} pixbuf_t;
```

Allocation of pixbuf_t's is done by the functions in graphics_layers.c and graphics_layers.h.

In addition to the composited SDRAM layer, 3 other layers are supported. An overview of the usage of the layering system is as follows:

1. Initialize the composited layer by calling *graphics_init()*
    a. This will allocate the memory for the composited layer (final buffer), and pass the base address of it to the VIDEO_FB_STREAMER hardware component.
    b. A pointer to this layer (pixbuf) is returned when calling *graphics_get_final_buffer().*
2. Initialize other layers by calling *graphics_layer_add()*
3. Write graphics to any of the above layers by passing the pixbuf_t* to various drawing commands. These pointers can be acquired by calling the *graphics_layer_get()* function.
4. Copy each of the other layers to the composited layer
5. Call the ALT_CI_CI_FRAME_DONE instruction to mark the frame as done and ready for output.

The C-API exposes a number of graphical primitives. While some primitives are built purely in hardware, others are a combination of hardware and software.

Primitive / Operation	Function Signature	Software or Hardware	Comments
graphics_draw_pixel	`void graphics_draw_pixel(pixbuf_t *pixbuf, int x, int y, unsigned char color)`	Hardware	Simple memory copy into a memory location.
graphics_draw_rectangle	`void graphics_draw_rectangle(pixbuf_t *pixbuf, int x1, int y1, int x2, int y2, unsigned char color)`	Hardware	Hardware Accelerated.
graphics_draw_line	`void graphics_draw_line(pixbuf_t *pixbuf, int x1, int y1, int x2, int y2, unsigned char color)`	Hardware	Uses Bresenham's algorithm, no floating-point number. Gracefully handles coordinates beyond the edge of the screen.
graphics_clear_screen	`void graphics_clear_screen()`	Hardware	Draws a rectangle in colour 0x00 to fill the screen.
graphics_clear_buffer	`void graphics_clear_buffer(pixbuf_t *pixbuf)`	Hardware	Draws a rectangle into a pixbuffer (of dimensions equalling the size of the pixbuffer) with colour 0x00.
draw_letter	`void draw_letter(pixbuf_t *pixbuf, int y1, int x1, int color, int pixel_size, char* letter)`	Software	Calls multiple rectangle commands to draw a character. Each "pixel" of the font is a rectangle of size *pixel_size*.
print2screen	`void print2screen(pixbuf_t *pixbuf, int x1, int y1, int color, int pixel_size, char* string)`	Software	Calls the draw_letter function for each letter in the string. Does not support printf format delimiters, calling sprintf to prepare the output string may be required.
graphics_draw_circle	`void graphics_draw_circle(pixbuf_t *pixbuf, int cx, int cy, int radius, int color, int filled)`	Hardware/ Software	Drawing an unfilled circle is done entirely in hardware using Bresenham's circle algorithm and is very fast. Drawing a filled circle with *filled<0* creates concentric circles of radius *radius* to radius *radius * filled*. This

			effect is fast, but does not produce entirely filled circles due to interference patterns caused by mathematical discretization of the pixel coordinates. The effect was left in because it is fast and suitable for explosion graphics. Drawing a filled circle with *filled == 1* draws a filled circle, but the filling of the circle is accomplished by drawing lines from the centre point radiating out to all pixels plotted.
graphics_draw_rounded_rect	`void graphics_draw_rounded_rect(pixbuf_t *pixbuf, int x1, int y1, int x2, int y2, int radius, int filled, unsigned char color)`	Software	Calls the rectangle and filled circle primitives.
graphics_draw_triangle	`void graphics_draw_triangle(pixbuf_t *pixbuf, int x1, int y1, int x2, int y2, int x3, int y3, int filled, int color)`	Software	If *filled==0*, draws 3 lines. Otherwise, employs a complex “line-sweep” algorithm to fill the triangle. Depending on the size of the filled triangle, this algorithm can be very slow.
copy_buffer_area	`void copy_buffer_area(pixbuf_t *source, pixbuf_t *dest, rect_t *source_area, point_t *dest_offset)`	Hardware	Copy part of a pixel buffer to another pixel buffer in memory.
copy_buffer_area_transparent	`void copy_buffer_area_transparent(pixbuf_t *source, pixbuf_t *dest, rect_t *source_area, point_t *dest_offset, unsigned char t_color)`	Hardware	Copy part of a pixel buffer to another pixel buffer in memory. Colors matching t_color will not be copied, leaving whatever color was previously present at that location.

Inspection of *graphics_commands.c* shows which hardware registers each of the above operations modify and how the custom hardware instructions are called. Note that the C-API abstracts away the need to write directly to the hardware registers.

The software overall design of a project using our graphics system would be along the lines of the following diagram.

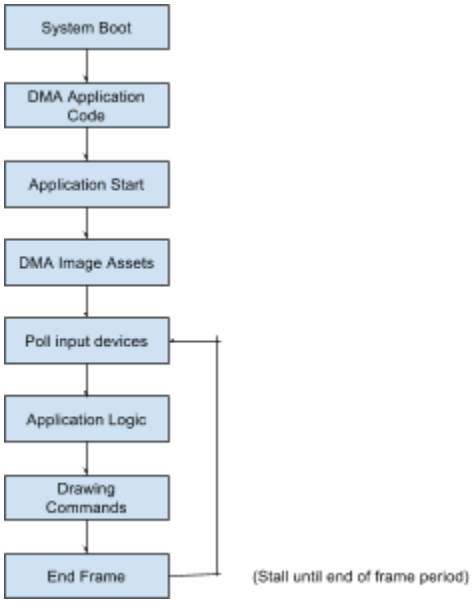
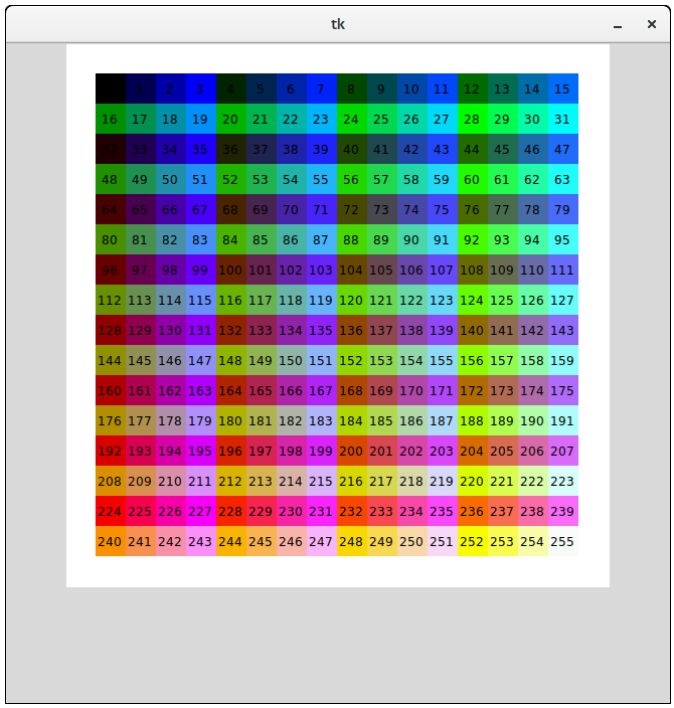


Image Loading Tools

We have also developed some tools for making image loading easier on our board. We wrote a python script called *image_converter.py* to read in an image file, and output a .BMP file and a *.pal* binary palette file. The input image is read in, and posterized to 256 colours. The accompanying palette file is produced so the picture can be reproduced faithfully on our board.

As well, we wrote a small utility to visually display the contents of a palette file. A screenshot of this utility is shown below.



Genesis Controller Logic

We wrote a VHDL block to expose the state of the Genesis controller as a memory-mapped register. This was possible because of the very simple design of the Genesis controller internals.

The VHDL block reads the controller pins when the Select pin on the controller is asserted to high and low to determine which buttons are pressed. The schematics of the controller show that we can gather the following information from it.

Pin	Function (Select low)	Function (Select high)
1	Up	Up
2	Down	Down
3	Logic low	Left
4	Logic low	Right
5	Power (+5 V)	Power (+5 V)
6	A button	B button
7	Select signal	Select signal
8	Ground	Ground

9	Start button	C button

Our project pulses the select line every 60ms to read the controller's button value.

A C-API is exposed to the user. An example usage is shown below. More detailed information can be found in our Application Note entitled "Sega Genesis Controller Interfacing."

```
#include <stdio.h>
#include <system.h>
#include <genesis.h>
int main(void)
{
// Initialize the Genesis controller interface
genesis_open_dev(GENESIS_0_NAME);
genesis_controller_t player1, player2;
while (1)
{
// Poll the status of each Genesis controller
player1 = genesis_get(GENESIS_PLAYER_1);
player2 = genesis_get(GENESIS_PLAYER_2);
// Check which buttons are pressed on controller 1
if (player1.up){
printf("1 Up was pressed\n");
}
if (player1.down){
printf("1 Down was pressed\n");
}
if (player1.left){
printf("1 Left was pressed\n");
}
if (player1.right){
printf("1 Right was pressed\n");
}
if (player1.a){
printf("1 A was pressed\n");
}
7if (player1.b){
printf("1 B was pressed\n");
}
if (player1.c){
printf("1 C was pressed\n");
}
if (player1.start){
printf("1 Start was pressed\n");
}
// Check which buttons are pressed on controller 2
if (player2.up){
printf("2 Up was pressed\n");
}
if (player2.down){
printf("2 Down was pressed\n");
}
if (player2.left){
printf("2 Left was pressed\n");
}
if (player2.right){
printf("2 Right was pressed\n");
}
if (player2.a){
printf("2 A was pressed\n");
}
if (player2.b){
```

```
printf("2 B was pressed\n");
}
if (player2.c){
printf("2 C was pressed\n");
}
if (player2.start){
printf("2 Start was pressed\n");
}
}
return 0;
}
```

Test Plan

The test plan for our project involved incrementally building up and testing each component of the hardware within the FPGA. Each specialized test environment (with only some hardware implemented) will have special test software that will exercise that particular component. This particular methodology is consistent with the “Bottom-Up” integration testing methodology introduced in ECE 322.

The order of our tests is as follows:

1. Test the VGA signal generation block.
    a. Verify that the VGA signal generation block can drive a monitor and display a test pattern.
2. Test that the VGA signal generation block chosen can display any arbitrary image, not just the test pattern supplied.
    a. Again, this is to ensure that the VGA generation block works as advertised and will meet our needs.
3. Build and test the palette shifter logic.
    a. Modify the data stored in the SRAM. Prior to this step, each byte in the SRAM framebuffer will contain an RGB233 value corresponding to the colour displayed. After this, we aim to hard-code the colour palette into the palette shifter. With this test, we aim to verify that the block works appropriately within timing constraints.
    b. Once a hard-coded pallette shifter is verified to be working, we will implement the loading of pre-coded colour palettes from the SRAM into the palette shifter block. For symmetry, we will also allow the palette shifter block to write palettes into the SRAM. This will allow us to set and test the palette shifter’s capability to load and store palettes without needing to build a separate qsys block that “drives” the palette store operation.
4. Next, we will test the genesis controller logic.
    a. The genesis controller qsys block is quite simple. It polls the genesis controller frequently (toggling the select line in order to get reads of all buttons) and stores the value in an internal register.
    b. To complete these tests, we will build an architecture for the block called “test”. The test architecture will implement the controller block entity and mirror the avalon architecture, with the exception that a series of LEDs will be

pulsed in a predetermined pattern to verify correct operation of the controller read.

5. Next, we must stress-test the memory bandwidth constraints inherent to the system.
    a. For this testing phase, we will populate the “composited frame” region of the SDRAM.
    b. From there, we need to perform a hardware DMA with the appropriate soft hardware on the FPGA to verify our SRAM’s memory access timing constraints. This testing step will verify whether or not we can “outrun” the VGA display signals and populate the SRAM’s frame (portion-by-portion) during the VBI.
    c. After this testing is complete, we will know that the system is theoretically able to render and output 10 frames per second.
6. Next, we must verify that we can run our two NIOS/II cores with their appropriate memories.
    a. We must set up each NIOS/II’s memory map such that it can use the SDRAM for variable storage without overwriting the framebuffers stored in memory.
7. Once each NIOS/II is verified as being capable of running code without disrupting the framebuffers stored in SDRAM, we can begin to implement and test the custom vdp instructions.
8. Next, we can write a test dummy display program that verifies that our hardware is capable of displaying a static frame.
    a. This program will be further refined such that we test more of the features promised in the proposal document.
9. Once the hardware features are confirmed to be working, we will begin writing our demo application: pong. Pong will be tested for performance and will be extensively play tested to check for bugs.

Results of Experiments and Characterization

Using a simple C-program to directly write a pixel color from SDRAM to SRAM gave a ballpark figure in regards to memory performance in software. Writing the entirety of a frame took 3 to 5 seconds with a Nios II-e, or only 1 second on a Nios II-s with hardware multipliers. Several instructions are required to perform a memory write from software, whereas performing these writes via direct memory access would allow for two pixels to be written per clock cycle with no processor overhead. This was the baseline that we used to determine our project’s acceleration capabilities.

There is a minimal latency of about eight clock cycles to write all pixels in the SRAM to the display. This is virtually instantaneous and presents no issues for us going forward. We are currently experiencing minor flickering issues in the test pattern which will be trivial to fix with some slight modifications to our PLL configuration.

Although the graphical primitives employ Bresenham’s line algorithms, we found that the peer-reviewed citations within [11] adequately show that the algorithms are correct and

performant. As a result, we did not perform numerical simulations of Bresenham's algorithms.

Numerical simulation of the algorithms would entail verification of correctness, or verification of performance. During our development of the VHDL code implementing the algorithms, we were able to visually distinguish whether the result was correct: if the state machine did not terminate, the CPU would lock up and nothing at all would be displayed. When testing the line algorithm, we were able to distinguish correctness in the "stair-stepping" by drawing lines at 45 degrees (for each horizontal pixel incremented/decremented, a vertical pixel was incremented/decremented) as well as verifying that the end coordinates of the line did not have off-by-one errors.

Our performance figures in the *datasheet* section of this report show that our system is highly performant. The primitives were timed using an Altera Timestamp timer. The 60fps and 30fps drawing deadlines were also found using the timestamp timer.

There is no need for calibration, due to lack of analog inputs to calibrate against.

Safety

Our system consists of the standard Altera DE2 board, in addition to an adapter board to interface with two sega genesis controllers. As such, the primary physical safety concerns of our implementation are those of the DE2 board. Namely, users of our product should be careful around electrical outlets, power cords, and avoid tasting the solder joints. The board can reliably operate at 0 to 85 degrees Celsius. At maximum load, the DE2 will draw 1.3A and 9V, for a maximum power consumption of 11.7 Watts.

Regulatory and Society

Our project does not make use of any personal information from its users, and as such requires minimal regulation. There is no internet connectivity involved in our project, and we are not subject to the typical concerns associated with networked "internet of things" devices. All user interaction will be performed through the use of Sega Genesis controllers in order to manipulate graphics, and so the ability for a user hack our device is largely minimized. It could be possible for a user with sufficient time and effort to interface with one of the DB9 ports to send inputs in such a way that memory could be altered in a way to allow for arbitrary code execution, as has been demonstrated on the SNES console in the past [8]. However, as we implemented a simple Pong game, the level of possible inputs and changes in memory relative to a complex SNES game reduces this risk of hacking. If the platform were to be adopted by developers, video games would be subject to the same regulations regarding subject matter that Sony or Nintendo would have to follow such as implementing "Licensed Developer" schemes (such as "Nintendo Seal of Quality") to restrict the content run on the system technically and artistically.

Environmental Impact

Our device is not RoHS compliant because the SEGA genesis controllers contain leaded solder. Unleaded solders could be used in place of this solder. The DE2 board used with our project is RoHS compliant.

In addition to the issue above regarding dangerous materials, our device would be characterized as a leisure device which could be identified as unnecessary for human survival, and therefore uses more resources to produce the power required to operate the device than would otherwise be necessary.

Sustainability

Our project draws 0.45 Amps at 8.93 V for 4.02 W of power.

$$P = I \times V$$
$$P = 0.45A \times 8.93V = 4.02W$$

The average monthly cost of power in Edmonton over the last 12 months is 5.37 cents per kilowatt-hour of power [9]. Assuming that our project is operating at maximum power draw for twenty-four hours a day, seven days a week, this results in an estimated cost of $1.89 in electricity per year for 35.22kWh of power.

$$E = P \times t$$
$$E = 4.02W \times 24h \times 365 = 35215.2Wh = 35.22kWh$$

$$Cost = E \times Price$$
$$Cost = 35.22kWh \times 0.0537 = \$1.89$$

According to Environment Canada, Albertan CO2 emissions for utilizing electricity are 820g of CO2 per kWh of electricity used. [10] For non-stop all day usage over the course of a year, our project will release 30.1 kg of CO2 into the environment. (820g * 36.72 / 1000 = 30.1 kg)

Due to the nature of our project, there are no idle or sleep states, it is always active and draws the same amount of power during operation.

References / Citations

[1]S. Larson, "VGA Controller (VHDL) - Logic - eewiki", *Eewiki.net*, 2016. [Online]. Available: https://eewiki.net/pages/viewpage.action?pageId=15925278. [Accessed: 18- Jan- 2016].

[2] Excamera.com, "Gameduino: a game adapter for microcontrollers — excamera", 2016. [Online]. Available: http://excamera.com/sphinx/gameduino/. [Accessed: 18- Jan- 2016].

[3] University of Utah - CS/EE 3710, "Computer Design Lab -- Lab 3, VGA", 2016. [Online]. Available: http://www.eng.utah.edu/~cs3710/labs/VGA.pdf. [Accessed: 18- Jan- 2016].

[4] ECE 448 - FPGA and ASIC Design with VHDL, "ECE 448 - VGA Display Part 1", 2016. [Online]. Available: http://ece.gmu.edu/coursewebpages/ECE/ECE448/S13/viewgraphs/ECE448_lecture7_VGA_1.pdf. [Accessed: 18- Jan- 2016].

[5]C. Rosenberg, "Sega Six Button Controller Hardware Info", *Cs.cmu.edu*, 1996. [Online]. Available: https://www.cs.cmu.edu/~chuck/infopg/segasix.txt. [Accessed: 18- Jan- 2016].

[6]"27. Video Sync Generator and Pixel Converter Cores", in *Altera Embedded IP User Guide*, 1st ed., 2016.

[7] Opencores.org, "VGA/LCD Controller :: Overview :: OpenCores", 2016. [Online]. Available: http://opencores.org/project,vga_lcd. [Accessed: 18- Jan- 2016].

[8] Arstechnica.com “How a Game Playing Robot Coded Super Mario Maker onto an SNES Live on Stage” [Online]. Available: http://arstechnica.com/gaming/2016/01/how-a-game-playing-robot-coded-super-mario-maker-onto-an-snes-live-on-stage/1/

[9] Epcor.com “Residential Rates and Fees” http://www.epcor.com/power-natural-gas/regulated-rate-option/Pages/residential-rates.apx

[10] https://www.bullfrogpower.com/wp-content/uploads/2015/09/2015_bullfrog_power_electricity_emission_calculator.pdf

[11] J. Bresenham, "Pixel-processing fundamentals," in *IEEE Computer Graphics and Applications*, vol. 16, no. 1, pp. 74-82, Jan 1996.
doi: 10.1109/38.481626,
URL:
http://ieeexplore.ieee.org.login.ezproxy.library.ualberta.ca/stamp/stamp.jsp?tp=&arnumber=481626&isnumber=10283

Appendices

Quick Start Guide

Hardware Setup

Required Components:

1. DE2 Board
2. “Sega Genesis Controller Adapter” PCB
3. 40-pin Ribbon Cable
4. 2x Sega Genesis Controller
5. VGA Monitor
6. VGA Cable
7. SD Card (.sof only)

Prepare SD Card (.sof only):

1. Create a FAT-16 formatted SD Card - you must use an old card that is not SDHC
2. Copy **small.bmp** and **small.pal** to the SD Card

Setup Instructions:

1. Use ribbon cable to connect DE2’s “GPIO 1” to “Sega Genesis Controller Adapter”
2. Plug Sega Genesis Controller into each port on “Sega Genesis Controller Adapter”
3. Connect DE2 to VGA Monitor with VGA Cable
4. (.sof only) Insert SD Card into DE2

Programming Board:

- **Volatile:** Run **program_volatile.sh**
- **Non-Volatile:** Run **program_nv.sh**

Demo Instructions

DE2 Buttons:

- KEY0: Reset

Genesis Controller Buttons:

- START: Next Demo
- [PONG] UP / DOWN: Move Pong paddle up / down
- [PONG] B + LEFT / RIGHT: Move Pong paddle left / right
- [PONG] A: Turn on / off “trajectory display”
- [PONG] A + B: Deflect ball at maximum speed towards opponent
- [PONG] A + B + C: Enable “the wall”
- [PONG] B + C: Gradually slow down opponent

Controls

ALL	**START**	Next demo
PONG	**UP / DOWN**	Move paddle Up / Down
PONG	**B + LEFT / RIGHT**	Move paddle Left / Right
PONG	**A**	Turn on guides
PONG	**A + B**	Launch ball at max speed
PONG	**A + B + C**	Enable “the wall”
PONG	**B + C**	Slow down opponent

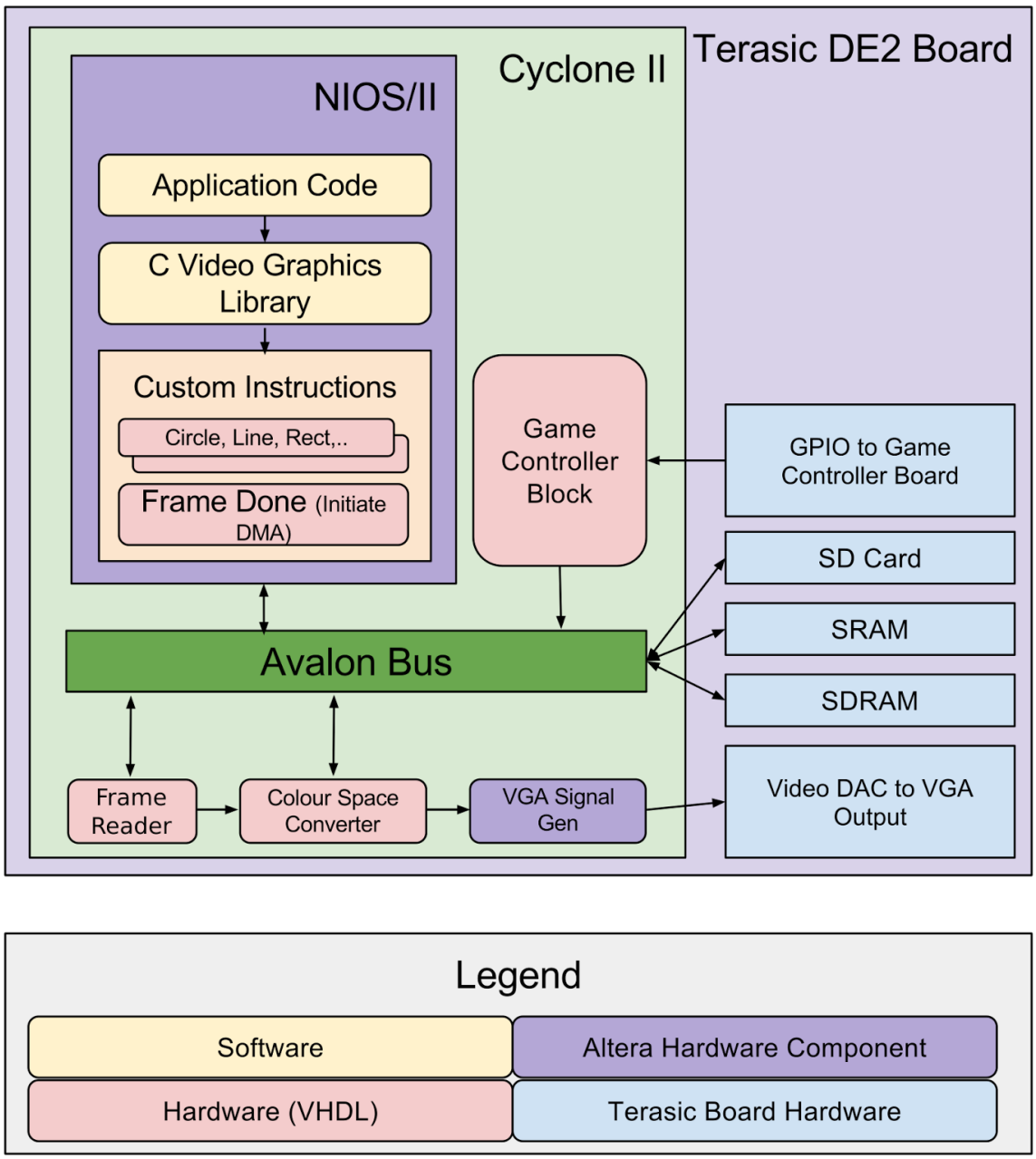
Note: If the Pong game seems to be slower than normal, reboot the system.

Future Work

A number of things could be done in the future to extend or better make use of our platform. For example, different applications could be written to make use of the platform - perhaps other games like Tetris or Air Hockey. As well, audio support could be added to the platform for an integrated console experience. In addition, the code could be ported to a newer development board containing more RAM in a different arrangement allowing for a higher frame resolution and removal of the 256 colour output limit. The graphics primitives supported in hardware could also be extended to support more advanced operations such as rotation, or drawing dotted lines instead of solid lines. Another application for our platform would be for integration of a live video stream into the background layer, allowing for graphics to be dynamically overlaid. This could assist in rear-view camera applications in vehicles, where guidelines could be superimposed over the video feed to assist the driver.

Hardware Documentation

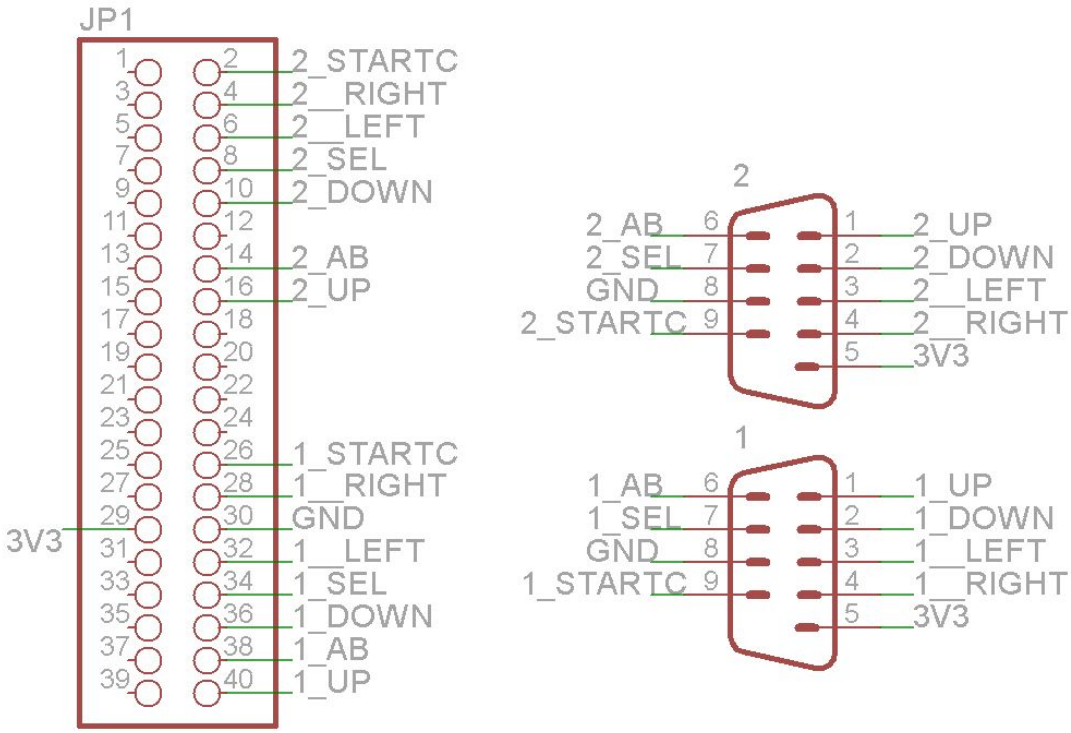
Hardware Block Diagram



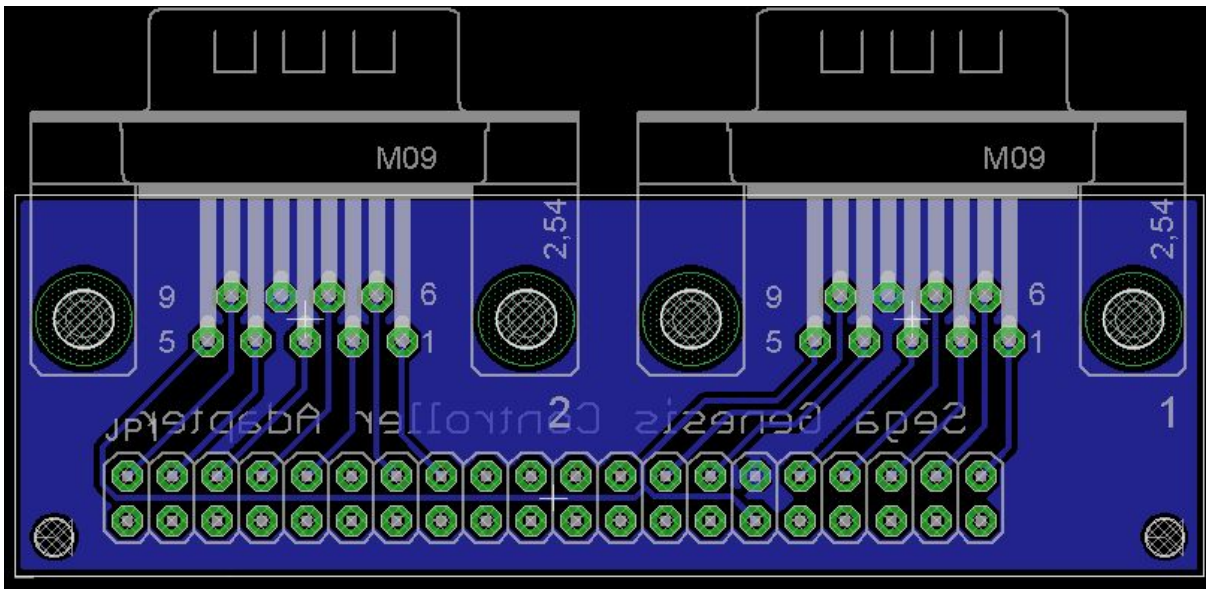
Genesis Adapter Board Documentation

In accordance to the Genesis controller specification outlined in [5], we have designed an adapter board that will allow us to connect the Sega Genesis controller to the FPGA's GPIO pins. We have designed this hardware using a free version of Eagle.

Schematic



Layout



Source Code

Our source code can be found on GitHub:
https://github.com/stephenjust/de2-video-processor

There is also a copy of the source code located at:
https://www.ualberta.ca/~delliott/local/ece492/projects/2016w/g6_graphics_system/de2-video-processor-master.zip

All of the code included with the project has been compiled and tested by various methods using a DE2 board. Note that some of the development tools may require extra Python libraries.

All VHDL components are connected via Qsys. All software projects include the video_system_graphics_library project as a dependency.

File Listing

File	Description
de2_video_processor.qpf	Quartus project file
de2_video_processor.qsf	Quartus project file
de2_video_processor.sdc	Timing constraints file
de2_video_processor_system.qsys	QSYS system definition
default_palette.mif	Default colour palette memory
default_palette_ega.mif	Example palette memory with only the EGA colour palette
ip/ci_copy_rect/HDL/ci_copy_rect.vhd	Custom instruction to copy a rectangular area of (discontinuous) memory
ip/ci_copy_rect/ci_copy_rect_hw.tcl	Qsys component definition
ip/ci_draw_circ/HDL/ci_draw_circ.vhd	Custom instruction to draw the outline of a circle to a pixel buffer
ip/ci_draw_circ/ci_draw_circ_hw.tcl	Qsys component definition
ip/ci_draw_line/ci_draw_line.vhd	Custom instruction to draw a line to a pixel buffer
ip/ci_draw_line/ci_draw_line_hw.tcl	Qsys component definition
ip/ci_draw_rect/ci_draw_rect.vhd	Custom instruction to draw a rectangle to a pixel buffer
ip/ci_draw_rect/ci_draw_rect_hw.tcl	Qsys component definition

ip/ci_frame_done/HDL/ci_frame_done.vhd	Custom instruction to trigger frame data copy from SDRAM to SRAM
ip/ci_frame_done/ci_frame_done_hw.tcl	Qsys component definition
ip/colour_space_converter/HDL/colour_space_converter.vhd	Palette decoder component
ip/colour_space_converter/colour_palette_shifter_hw.tcl	Qsys component definition
ip/common/avalon.vhd	VHDL package including a couple of Avalon-bus helper components
ip/common/avalon/avalon_copy_sequential.vhd	Component to copy a block of memory to a destination, at 8 bits per clock.
ip/common/avalon/avalon_copy_sequential_16.vhd	Component to copy a block of memory to a destination, at 16 bits per clock.
ip/common/avalon/avalon_write_sequential.vhd	Component to write a byte to a continuous segment of memory, 16 bits per clock.
ip/common/geometry.vhd	VHDL package including geometry-related helper functions and struct (record) definitions.
ip/genesis/HAL/inc/genesis.h	Header file for Genesis driver
ip/genesis/HAL/src/component.mk	Makefile for Genesis driver
ip/genesis/HAL/src/genesis.c	Source file for Genesis driver
ip/genesis/HDL/genesis.vhd	Genesis controller interface component
ip/genesis/genesis_hw.tcl	Qsys component definition
ip/genesis/genesis_sw.tcl	Qsys software driver definition
ip/video_fb_streamer/HDL/video_fb_dma_manager.vhd	Component that handles memory transfers to/from SDRAM and SRAM for the output pipeline
ip/video_fb_streamer/HDL/video_fb_fifo.vhd	Component to take two pixels in at a time, and output one pixel at a time on a second clock
ip/video_fb_streamer/HDL/video_fb_sdram_reader.vhd	Component to read SDRAM buffer in bursts
ip/video_fb_streamer/HDL/video_fb_streamer.vhd	Component to manage moving pixel data from SDRAM to SRAM, and then streaming that data to the output pipeline
ip/video_fb_streamer/video_fb_streamer_hw.tcl	Qsys component definition
release_files/program_nv.sh	Script to program EPSC and Flash from binaries
release_files/program_volatile.sh	Script to program FPGA over JTAG and launch code from elf
software-tools/RGB323toRGB565_mif_palette_generator.js	Script to generate the default_palette.mif file
software-tools/c_palette_generators.js	
software-tools/image_converter.py	Script to convert image files to 8-bit paletted

	bitmaps, and to generate the palette files.
software-tools/palette_shower.py	Script to read palette file and show colours and their indexes.
software/Pong/main.c	Main program logic for final demo application
software/Pong/pong_graphics.{c,h}	Graphics helpers for Pong game
software/Pong/pong_helpers.{c,h}	Game logic helpers for Pong game
software/benchmark_test/main.c	Benchmarking program to get performance of graphics operations
software/compositing_test/main.c	Program to give an example of how to use layering
software/de2_video_processor/hello_world.c	Sample program to test Genesis controller input
software/draw_images/main.c	Sample program to test bitmap drawing
software/graphics_and_font_test/main.c	Sample program to test graphics primitives
software/line_test/main.c	Sample program to test line drawing
software/rectangle_test/main.c	Sample program to test rectangle drawing
software/sdram_tearing/main.c	Sample program to test video output tearing
software/sdram_test_pattern/main.c	Sample program to test frame swapping from SDRAM to SRAM
software/sram_simple_geometry/main.c	Sample program to test for video glitches
software/sram_test_pattern/main.c	Sample program to test SRAM buffer to video output
software/sram_test_pattern_palette_switch/hello_world.c	Sample program to test the palette switcher
software/video_system_bsp/settings.bsp	Common BSP project
software/video_system_graphics_library/efsl/*	EFSL library to read files from SD card
software/video_system_graphics_library/flash_ops.{c,h}	Helper functions to read images and palettes from flash
software/video_system_graphics_library/graphics_commands.{c,h}	Main collection of graphics helper functions
software/video_system_graphics_library/graphics_defs.h	Common graphics library definitions
software/video_system_graphics_library/graphics_layers.{c,h}	Helper functions to handle graphics layers
software/video_system_graphics_library/palettes.{c,h}	Helper functions to handle colour palettes
software/video_system_graphics_library/sdcard_ops.{c,h}	Helper functions to read images and palettes from SD card
src/de2_video_processor.vhd	VHDL Top-Level