Basic Components of a Game Console

- Graphics / Video Output
- Audio Output
- Human Interface Device (Controller)
- Game program on separate media

My goal is to design and build a game console with these basic components.
Original Project Goals

- Project aimed to be:
  - Simple (by today’s standards) 2D Game Console
  - Low power, ARM and FPGA based design
  - Comparable to GBA or SNES in video capability…
  - …except for higher resolution
  - ‘Nintendo-like’ video design: tiled backgrounds, sprites, paletted graphics, etc.
  - Full 44 kHz, stereo audio support
  - Possible support for existing NES controller
Results

- I was more or less able to accomplish most of my goals, although there are a few bugs.
- Final status of video controller:
  - 640x480 VGA resolution, 16-bit color depth
  - LPC2148 ARM Microcontroller @ ~59 MHz
  - Spartan-3e 500k gate FPGA @ 150 MHz (was 190)
  - 16 MB Video RAM
  - Two 32x32 pixel scrolling tiled BGs (was three)
  - 16 32x32 pixel sprites
  - 44.1 KHz 8-bit stereo audio
  - SD card storage for audio and graphics data
  - NES controller input
  - BG Transparency, adjustable opacity, and brightness
PCB Layout
Video Controller Design

- Features based off GBA video controller
- GBA Graphics features:
  - Scrollable Tiled Backgrounds
  - Palette Graphics
  - Sprites
  - Background/Sprite Rotation and Scaling
  - Background/Sprite Transparency
  - Various Video Modes
  - Many other minor features
Video Controller Design

- Was not able to replicate all of the GBA’s features due to time, complexity, and hardware limitations
  - Rotation and Scaling
  - BG Prioritization
  - Horizontal/Vertical Sprite Flipping
  - And many other things…
- My video controller surpasses the GBA in other areas though…
Video Controller Design

- Game Console Video Controller Features
  - 640x480 Pixel Resolution (VGA) @ 60 Hz
  - 16-bit Color (RGB555 Format), 32768 Colors
  - 16 Mbytes Video Memory
  - 2 Scrollable Tiled BGs of 32x32 Pixel Tile Size
  - Paletted or Non-Paletted BGs and Sprites
  - Tile Maps of 32x16 Tiles (1024x512 Pixels)
  - 16 32x32 Pixel Sprites
  - BG Transparency and Adjustable Opacity
  - Brightness Control
Tiled Backgrounds

- Tiled Backgrounds:
- Commonly used in 2D video game consoles
- Image made up of individual tiles
- Allows few tiles consuming only a small amount of video memory to fill an entire screen

Tiled backgrounds require a tile map in order to place tiles in their proper position on the screen. Tile maps themselves can consume considerable space depending on tile size and tile map size.
Paletted Graphics

- The palette is a color lookup table (LUT).
- When paletted graphics are used, each pixel of graphics data is a pointer to a color in the LUT.
- The advantage to palettes is being able to change colors on the screen by only changing the palette instead of having to change every pixel on the screen.
Video Modes

Bitmap Background

Tiled Background

Tile Background with Palette
Video Controller Design

- Video controller implemented on the FPGA.
- Design was written in VHDL.
  - VHSIC Hardware Description Language
    - Very High Speed Integrated Circuit (VHSIC)
- Mostly written at the Register Transfer Level (RTL).
  - Avoided writing counters, adders, comparators, etc.
    - Simply used VHDL libraries for that…
Video Control Subsystem

Note: BG Map 3 is Sprite Layer

Bgmapoffset = 3?

No

Rasterpos < 704?

Yes

Calculate BG X and Y offset

Place Tilemap address on bus

Set memory clock signal high

Tilepixpos = (yoff mod 32) * 32

Tilepixoffset = xoff mod 32

Calculate tilepixpos and tilepixoffset

Place tiledata address on bus

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Is there a sprite to be drawn?

No

address = Sprmoffset + (sprtlenum * 1024) + (hpxclock - sprposx) * 32

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Retrieve Sprite X Pos

Place sprite address on bus

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Note: BG Map 3 is Sprite Layer

Bgmapoffset = 3?

Yes

Rasterpos < 704?

No

Calculate BG X and Y offset

Place Tilemap address on bus

Set memory clock signal high

Tilepixpos = (yoff mod 32) * 32

Tilepixoffset = xoff mod 32

Calculate tilepixpos and tilepixoffset

Place tiledata address on bus

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Is there a sprite to be drawn?

No

address = Sprmoffset + (sprtlenum * 1024) + (hpxclock - sprposx) * 32

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Calculate BG X and Y offset

Place Tilemap address on bus

Set memory clock signal high

Tilepixpos = (yoff mod 32) * 32

Tilepixoffset = xoff mod 32

Calculate tilepixpos and tilepixoffset

Place tiledata address on bus

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Is there a sprite to be drawn?

No

address = Sprmoffset + (sprtlenum * 1024) + (hpxclock - sprposx) * 32

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Calculate BG X and Y offset

Place Tilemap address on bus

Set memory clock signal high

Tilepixpos = (yoff mod 32) * 32

Tilepixoffset = xoff mod 32

Calculate tilepixpos and tilepixoffset

Place tiledata address on bus

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Is there a sprite to be drawn?

No

address = Sprmoffset + (sprtlenum * 1024) + (hpxclock - sprposx) * 32

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Calculate BG X and Y offset

Place Tilemap address on bus

Set memory clock signal high

Tilepixpos = (yoff mod 32) * 32

Tilepixoffset = xoff mod 32

Calculate tilepixpos and tilepixoffset

Place tiledata address on bus

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Is there a sprite to be drawn?

No

address = Sprmoffset + (sprtlenum * 1024) + (hpxclock - sprposx) * 32

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Calculate BG X and Y offset

Place Tilemap address on bus

Set memory clock signal high

Tilepixpos = (yoff mod 32) * 32

Tilepixoffset = xoff mod 32

Calculate tilepixpos and tilepixoffset

Place tiledata address on bus

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Is there a sprite to be drawn?

No

address = Sprmoffset + (sprtlenum * 1024) + (hpxclock - sprposx) * 32

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Calculate BG X and Y offset

Place Tilemap address on bus

Set memory clock signal high

Tilepixpos = (yoff mod 32) * 32

Tilepixoffset = xoff mod 32

Calculate tilepixpos and tilepixoffset

Place tiledata address on bus

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Is there a sprite to be drawn?

No

address = Sprmoffset + (sprtlenum * 1024) + (hpxclock - sprposx) * 32

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Calculate BG X and Y offset

Place Tilemap address on bus

Set memory clock signal high

Tilepixpos = (yoff mod 32) * 32

Tilepixoffset = xoff mod 32

Calculate tilepixpos and tilepixoffset

Place tiledata address on bus

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Is there a sprite to be drawn?

No

address = Sprmoffset + (sprtlenum * 1024) + (hpxclock - sprposx) * 32

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Calculate BG X and Y offset

Place Tilemap address on bus

Set memory clock signal high

Tilepixpos = (yoff mod 32) * 32

Tilepixoffset = xoff mod 32

Calculate tilepixpos and tilepixoffset

Place tiledata address on bus

Set read signal high for memory controller

Set read signal low

Memory controller ready?

No

Yes

Is there a sprite to be drawn?

No

address = Sprmoffset + (sprtlenum * 1024) + (hpxclock - sprposx) * 32

Set read signal high for memory controller

Set read signal low

Memory controller ready?
Raster Buffer Control

- Controls what data goes into the raster buffers which will ultimately be displayed on the screen.
- The raster buffers store a single horizontal line of pixels.
- There are three raster buffers in this design. The first two are for storage while the raster line is rendered. The last one is what gets shown on the screen.
- They allow the time in between drawing periods to be utilized for rendering.
- This subsystem also handles transparency and opacity.
Each raster line is drawn into a raster buffer beginning at the start of the “back porch”.

Right before that, the buffer is copied into another raster buffer that is output on the screen.

These buffers are implemented as dual port RAM and a single memory element will not be written to and read from at the same time due to these operations occurring at different speeds.

The final raster buffer that gets drawn on the screen had already begun being written to before the first pixel of data is read from it.
Memory Controller

- Uses a state machine to control the signals to the external RAM.
- Behaves as its own separate design entity.
- Controlled by various input signals such as start, burst length, burst enable, halt, etc.
- Outputs a “datavalid” signal when a valid word of data is present on the data bus. The “datavalid” signal is the clock signal used for the raster buffer control subsystem.
Micron CellularRAM

- The dev board has a 16 MB RAM that serves as the main video memory.
- It is a PSRAM. It is actually DRAM but it can also be interfaced to like SRAM. It can operate at up to 80 MHz.
- It has several different modes of operation. The ones used in this design are…
  - Asynchronous Mode:
    This mode allows the RAM to be interfaced like any other SRAM. Its advantage is that it's very simple to use, but its disadvantage is that it is very slow (70 ns access time). (Used for writing)
  - Variable Latency Bursts:
    A clock signal is required for this mode. After the rising edge of the first clock cycle, there is a number of clock cycles that must be waited before the first word of data is valid. The subsequent words of data are then outputted on the rising clock edges. However, the RAM can halt the burst at any time in order to do a refresh. The memory controller must monitor a wait signal so that it knows when the data is valid. This mode offers the highest throughput, but is the most difficult to implement. (Used for reading)
I could not get the stereo audio DAC to work no matter what I did. Instead, I used a method of producing audio called PWM (pulse width modulation) that I came across when looking at some other homebrew game consoles.

PWM audio, at the minimum, requires no additional hardware other than a device that can produce a PWM signal. I simply wired the headphone jack to the FPGA. PWM works by averaging the power output through changing the duty cycle of each pulse. So, a duty cycle of 10% would be 10% of the total output power. The higher the PWM frequency, the better the averaging of the power output, which lowers noise. I made my own PWM on the FPGA. Since the FPGA can output higher clock rates than a microcontroller, I can achieve much better sound quality than all the other microcontroller implementations I have seen. The PWM frequency that I chose is 112.5 MHz. Divide this by 256, and then by 10, and you get the typical sound sampling rate of about 44.1 KHz. The 256 comes from the fact that it is 8-bit audio, and there are 10 PWM pulses per byte of audio.
Audio Buffer

- The audio buffer is implemented as a 4096x16-bit Block RAM in the FPGA.
- It behaves like a circular FIFO using two registers as input and output pointer addresses.
- There is an issue with keeping the buffer filled with data without overrunning the buffer.
- No signal going back to the microcontroller signaling if the buffer is low
- Made the best approximation on how fast to send data to the buffer
NES Controller Input

- NES controller interface is just an 8-bit shift register.
- A latch signal is sent to load the button states into each flip-flop.
- Then a clock signal is sent to shift the data out.
- The data is stored in the FPGA and the MCU uses the SPI bus to retrieve the button states from the FPGA on every frame.
**SD Card**

- The SD card slot is connected to the MCU via SPI bus.
- EFSL library was used with slight modification.
- The library handles SPI bus communication protocols and has FAT file system support.
The graphics data gets from the MCU to the FPGA via a 16-bit data/address bus, 5-bit command bus, and a bus clock.

Typically, the command and address is placed on the bus first. The bus clock is toggled. Then, the data is placed on the bus. The bus clock is toggled again. Currently, a reset command is required after each transfer.
Current bug list...

- Design is horribly unstable. The smallest change in the code can cause drastic graphics glitches.
- Several graphics glitches were covered up with resulted in more tiles being read than necessary.
- Top horizontal line is glitched.
- Several vertical pixels on left side are glitched.
- Sprites do not overlap top of screen.
- Opacity only looks decent with 2 BGs that have no transparent sections in them.
- MCU interface control needs some work.