Team PSX
Design Report

Anita Zhang
Arnob Mallick
Michael Rosen

October 16, 2013

Table of Contents

Overview

Our team's goal is to recreate the original Sony PlayStation (PSX) on the Virtex-7 VC707 FPGA Eval Board. We plan to implement all major components, including sound, video, controller input and game data from ROMs.

The PSX was originally planned as a new version of the SNES with CD-ROM input rather than the traditional cartridge. However, as Sony continued to work on the project, Nintendo, the makers of the SNES, grew increasingly frustrated with the new console and decided to pull support from the PlayStation. When Nintendo announced that they were breaking their sound-chip contract with Sony in favor of a foreign competitor, Sony decided to continue the project alone and create their own console for market. After a few year, the PlayStation was finally released, in December 1994 for Japan and in September 1995 for the US. The PSX contained a MIPS I, 32-bit CPU with 2 co-processors with a 2MB RAM, a 2D GPU with 1MB VRAM, SPU capable of generating 24 voices and 512KB RAM, a CDROM from reading games, new controllers with 10 buttons and 2 slots for EEPROM memory cards.

We are attempting to reconstruct these components as close to the originals as possible. However, due to the period in which this console was made, the documentation freely available is quite limited. Thus, some components will not be identical to those of the original system. The original PSX connected to TVs set via Composite video; however, the VC707 board only has support for HDMI; so our implementation will use HDMi rather than Composite video for connecting to a TV/Monitor.

In the following sections, we describe our plan of action for implementing the PSX with a schedule and component breakdown. The following sections are each dedicated to a single component; describing it in technical detail and our implementation of it.

Plan of Action

TASK	DURATION (weeks)	9/16	9/23	10/07	10/14	10/21	10/28	11/04	11/11	11/18	11/25	12/02
HDMI												
- SPDIF	2											
- VIDEO	2											
- CONFIGURATION	2											
CPU	2											
GTE	2											
Controller	1											
Memory/DMA	2											
GPU	3											
MDEC	2											
SPU	2											
Debug & Consolidation	4											

Figure 1 *- Projected timeline (updated 10/16)*

Our current schedule is shown above. At the time of writing this we are in the week of October 14 and each member (represented by each color) is working on their respective components (research, writing, testing, etc.). We are still in the building phase, meaning that each component is still being worked on. Because of this we cannot completely integrate any two systems together.

In the coming weeks we will finish just enough of the components to be able to load the BIOS onto our system. Once that happens we will be able to do a more thorough test with instructions and other functionality. It will also be especially useful because we will have some form of video output and so we can start outputting errors to the screen.

On the topic of video, at some point in our schedule we will have to go back to handling HDMI. On and off each member has been working on getting HDMI to display video, sound, *anything*. Shifting away of the hardware interface we were trying to create, we will now be experimenting with the software interface provided by Analog Devices.

When the BIOS is successfully loaded, we will be entering our last stretch of the project. At that point the major components will be ready and done and all that is left are smaller components and testing. We expect to have a lot to debug, especially during and after integration time, so we left a large buffer of time to accommodate.

Major Components

The PSX, or our implementation of it, can be broken down into 9 primary parts. These are the CPU, GTE, GPU, MDEC, SPU, Controller Interface, Main Memory + DMA, ROMs/Game Data, HDMI/Audio-Video Interface. As seen in the schedule, we planned out times and members to work on each piece. The entire system for the original PSX and our implementation can be view in the diagrams below:

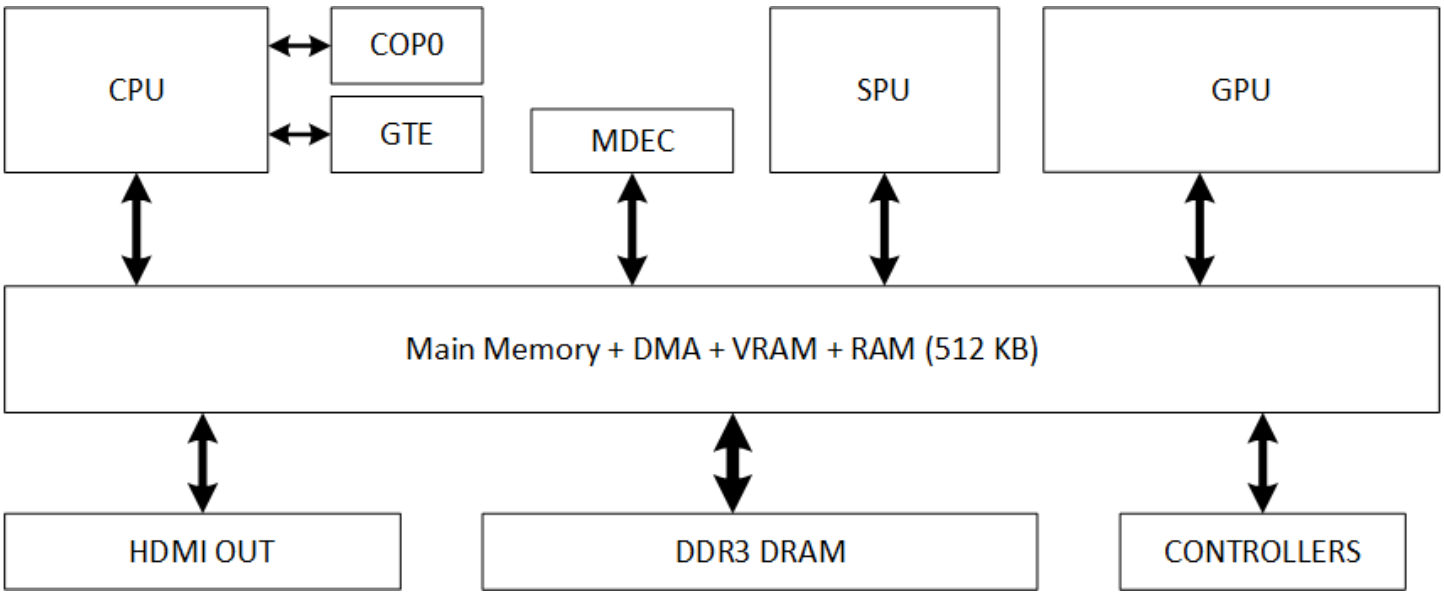

Figure 2 *- Original PSX System*

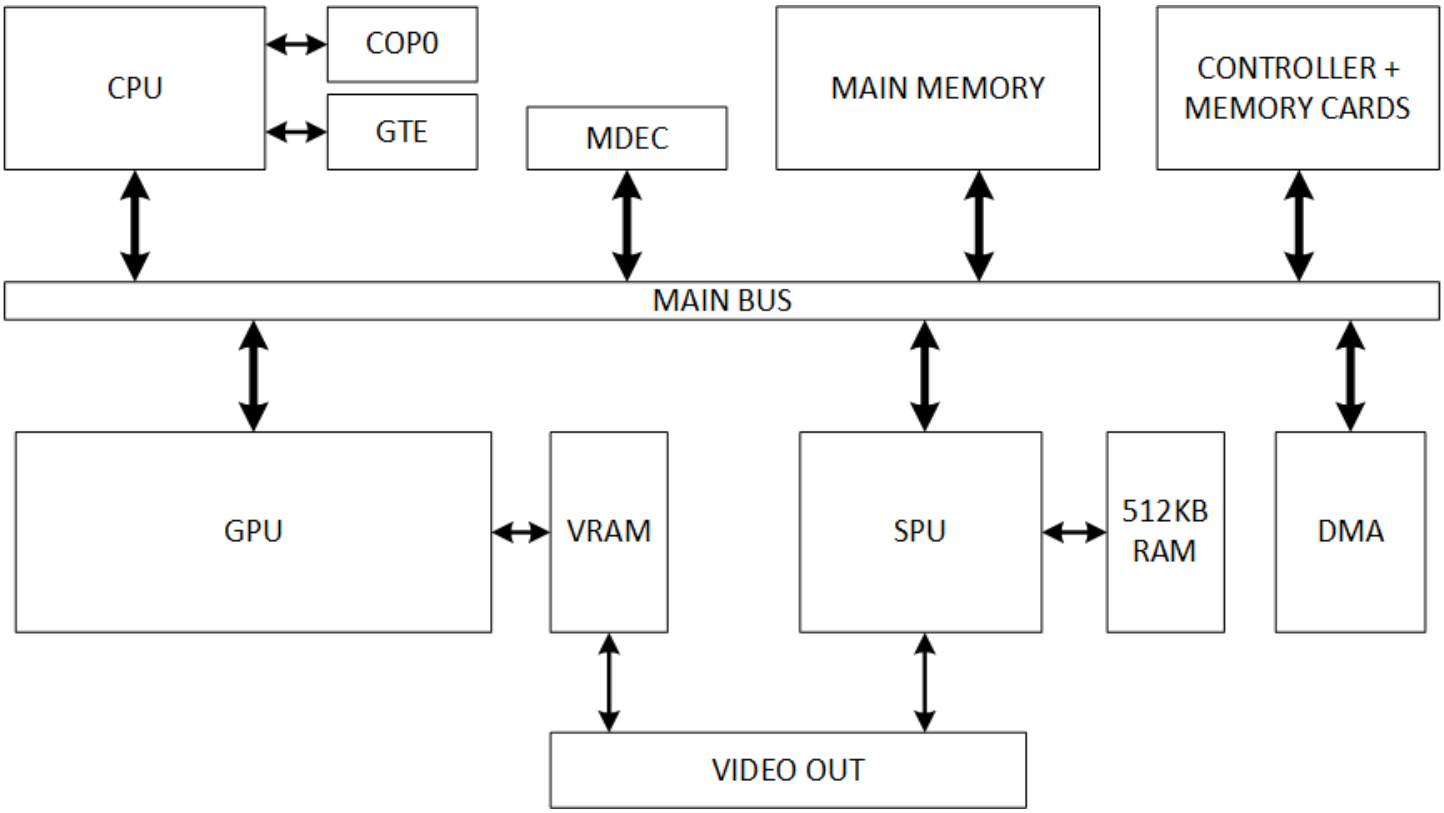

Figure 3 *- Our Implementation*

Our implementations quite similar. The major difference is that all memory is to be placed in the board's DDR3 DRAM. As such, there will be a module doubling as the main bus, DMA channels, VRAM the SPU's RAM. Most other units and their connections are the same as in the original system.

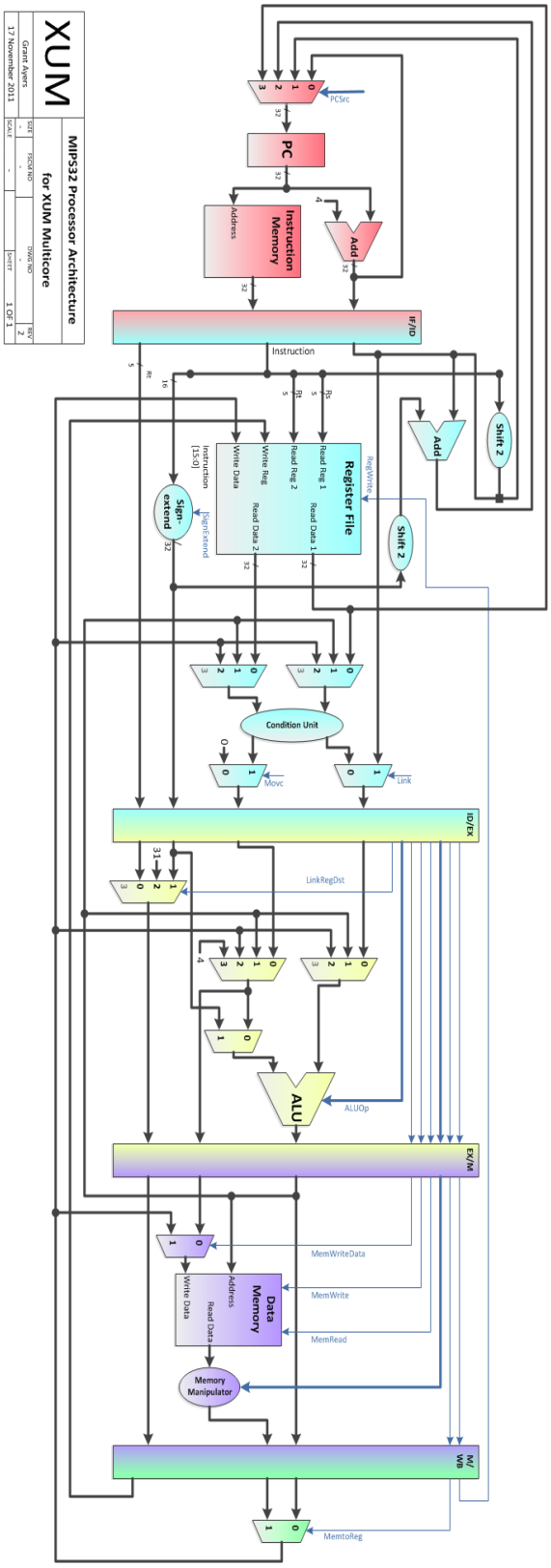

Figure 4 *- MIPS32 Processor Architecture*

CPU

The CPU we are using is an OpenCores MIPS CPU written by folks from the University of Utah. The schematic pictured at the end of this section is the one they created (XUM was the project/architecture they made the MIPS for). It is a fairly standard MIPS with a five stage pipeline, 5 general purpose hardware interrupts, one non-maskable hardware interrupt, data forwarding/hazard detection, and branch delay slots. In addition, it also implements a special memory handshake in which every memory access sends a request and waits for an acknowledge before moving on in the pipeline. We are fortunate enough to have the OpenCores CPU implement coprocessor 0, which handles all the exception detections in the CPU.

A classic RISC pipeline contains five pipeline stages for instruction fetching, decoding, execution, memory access, and writeback. The key points where each implementation of the MIPS differs comes from branch execution and data/control dependencies. In the OpenCores MIPS they have chosen to implement a single branch delay slot (as a standard MIPS would) instead of stalling. For performance reasons they have also implemented data forwarding. They based this on a "want" or "need" condition contained in each stage. For example, if data is available in the memory stage and the decode stage wants it, then the data will get forwarded. However if data is not available in the memory stage and decode needs it, then the decode stage will stall until the data is available. This same forwarding mechanism contains the hazard detection module.

One thing that may be a slowdown in their implementation is the memory handshake. In some systems it is enough to assume that the memory will be available on the next cycle. But for us we find the memory handshake is useful in case that assumption is not true. So even if it is a slowdown, for convenience and robustness purposes we are keeping it in.

It is important to note that the OpenCores MIPS does not implement virtual memory as the actual MIPS ISA specifies. However, the original PSX does not implement virtual memory with paging (as modern processors do nowadays). The memory architecture for the PSX, as documented by both fans and original developer documentation for the OS, works by separating virtual memory into three 2 MB segments: KUSEG, KSEG0, and KSEG1. Specifically, KUSEG is direct mapped and contains user and kernel data, KSEG0 is cacheable and is a mirror of the kernel and user data, and KSEG1 is the same as

KSEG0 but is not cacheable. Each region has a specific set of upper bits such that replacing them with a different set of upper bits will be the translation from virtual to physical memory. For example, in KSEG0 and KSEG1, replacing the 3 upper bits of the virtual address with “000” will give the physical address. Why the PSX does it like this? Who knows. But it does make the idea of implementing virtual memory a lot easier.

Coprocessor 0 is what handles virtual memory and exceptions in the MIPS processor. It contains several registers for this purpose, as well as a register for configuring additional coprocessors. As I said before, the OpenCores MIPS does not implement any virtual memory operations, but the subset it does implement is fully compatible with the MIPS specification. Exceptions can be divided into two categories: synchronous and asynchronous. Software interrupts/ exceptions appear synchronous in the program order and so no instructions after the interrupt should start until the interrupt has been processed. At the same time, all instructions in forward stages of the pipeline should complete before the interrupt processing is done. This is accomplished by stalling the pipeline in the stage that the exception appeared. Asynchronous interrupts have a little more leeway. The OpenCores MIPS Cop0 detects and handles asynchronous interrupts in the decode stage of the pipeline. Because of their asynchronous nature, forward instructions in the pipeline may either run to completion or get flushed and restarted. OpenCores has chosen to consistently allow forward instructions to run to completion. This makes it nicer to not have to handle instruction restarts, but this increases interrupt latency if, for example, a memory stall is in progress.

Other differences in the OpenCores MIPS may limit our performance when it comes to the actual PSX. As mentioned before, it does not implement virtual memory, and thus does not have a TLB implemented. Any form of caching available is also unavailable. Functionality-wise it should not be a problem, but for performance reasons we will have to implement them.

Another thing to note is that the PSX CPU is LSI’s LR33x30 implementation of MIPS. The only notable difference in implementation in the LR33x30 compared to the OpenCores MIPS is load scheduling, which, for now, we are disregarding. As I said before, it is still functional as is so certain things like scheduling will be disregarded for now. There is a certain lack of documentation missing about the LR33x30, but the datasheet that exists does not contain anything deviating greatly from a standard MIPS.

GTE

< Eventually insert image here >

The Geometry Transformation Engine is the heart of all 3D calculations in the PSX. Anything involving vector/matrix operations, perspective transformations, color equations, etc. are done on the GTE. It has about 20 instructions, with variations depending on the arguments, especially suited for this task and thus is much faster than a general purpose CPU for these operations. It also contains several special load/store instructions for accessing the 32 32-bit control registers and 32 32-bit data registers.

The GTE is mounted to the PSX’s MIPS as coprocessor 2. To program the GTE, the appropriate bits in the status register of Cop0 must be set in order to enable Cop2. One of the great features of MIPS instructions’ encoding is that the first 6 bits of the instruction (may) indicate which coprocessor it is issuing commands to. And so programming with GTE commands are just like running any other MIPS instruction, just that the decode of the first 6 bits will tell us if it is a coprocessor instruction or not. Then the CPU will pass the instruction along and the GTE will handle it.

As a programmer of the GTE the biggest restrictions are that GTE instructions should not be used in branch delay slots, nor in event handlers or interrupts. The CPU or GTE does not seem to enforce this behavior on the hardware side, so it is a risk the programmer will have to handle.

The control registers in the GTE contain all of the scale factors, offsets, rotation elements, etc needed for the various transformations. Although each register is 32-bits wide, most of them contain two 16-bit

registers in specified signed fixed point format (most of them are 1.3.12). The data registers contain color, vector values, sums, etc. as results of the various instructions. Some of them may contain two 16-bit registers as seen with the control registers, and some come in different fixed point formats (depending on the operation). All of this is detailed in the commands listing (not detailed here, but the overview is coming up).

Some (minimal) background on vector math: 3D coordinates are represented through a vector consisting of X, Y, and Z. The GTE considers two kinds of vectors: variable length and normal vectors (unit length) in order to describe direction and location in 3D space. Rotating vertices involves multiplying the vector of the vertex with a rotation matrix which is a 3x3 matrix with 3 normal orthogonal vectors. Rotation about any axis has a specific matrix with which to multiply by. And yes, the order of multiplication is important.

To make implementation easier for our team, a fantastic document detailing each instruction (reduced to arithmetic mathematical notation) was found. It contains the instruction, cycle count, line by line equations needed to perform the instruction, and which registers is reads and modifies during the operation. Obviously this is not how the PSX GTE really works, but it enables us to create a cycle accurate implementation (you can think of it as an emulator) of the GTE, with accurate modifications of registers. Yay!

For our project implementation, three things happen when a GTE instruction enters the GTE module. First we stall the CPU. GTE instructions are not pipelined (as far as documentation goes), so eventually we should modify this such that we only stall if the following instructions are other GTE instructions (to allow regular instructions to flow forward). Then we find out which instruction it is and load its cycle count into the counter. Third we match the current cycle to the sub-instruction for that instruction. On each cycle a different sub-instruction executes until the instruction is completely done. Then the counter is cleared and the stall is removed.

One of the challenges of implementing the GTE is the fixed point math and overflow handling. The command list does tell us which flags to modify in case a sub-instruction overflows, but there are also situations in which the result is output to a 44+ bit fixed point number and will eventually need to be stored back in a 32 bit fixed point register. How does rounding work for cases like that? How do we confirm that what we store back is what a real PSX would do? For testing we have found GTE test programs on the Internet meant to run on actual PSX hardware or emulators. Unfortunately they are not Verilog testbenches. The plan is to run those test programs on an emulator or the PlayStation we bought, and to capture the input and outputs. Verification after that will be easy to do.

0x0000_0000-0x0000_ffff	Kernel (64K)
0x0001_0000 0x001f_ffff	User Memory (1.9 Meg)
0x1f00_0000-0x1f00_ffff	Parallel Port (64K)
0x1f80_0000-0x1f80_03ff	Scratch Pad (1024 bytes)
0x1f80_1000-0x1f80_2fff	Hardware Registers (8K)
0x8000_0000 0x801f_ffff	Kernel and User Memory Mirror (2 Meg) Cached
0xa000_0000 0xa01f_ffff	Kernel and User Memory Mirror (2 Meg) Uncached
0xbfc0_0000-0xbfc7_ffff	BIOS (512K)

Figure 5 *- PSX Memory Map*

MAIN MEMORY/DMA

The PSX has a few memory components serving various purposes. The primary memory chips are Main memory, VRAM, and Sound Ram (SRAM). In addition there is also memory associated with the CDROM controller. The 32-bit addressing to Main Memory has several mirrors to the same physical address space. The top three bits of the address serve as a mapping to different memory related functions. If the address is mapped to KSEG0 with a ‘100’ prefix, then caching is enabled. Alternatively, memory that is mapped to KSEG1 with a ‘101’ prefix does not perform

any caching. Cop0 handles nearly all interactions with Main Memory. Refer back to the ‘CPU’ section for further elaboration on the workings of the PSX virtual memory and how the mirroring takes place. Main Memory also contains all of the memory-mapped I/O. All other computational units (GPU, SPU, MDEC, and CDROM) make use of registers located in Main Memory as well. This includes registers for all peripherals including controllers and memory cards. It also includes several control registers for various functions such as DMA, and timers. What would typically be the data cache in a standard MIPS CPU is instead used as a Fast RAM or “scratchpad” in the PSX. This region of memory can only be used for data can cannot be used to store code. The BIOS is stored in the last 512k of Main Memory. VRAM and Sound Ram are self-explanatory. They deal with their respective media outputs and processing. The GPU handles the frame buffer, texture pages and texture palettes in the VRAM.

	SIZE	DETAILS
Main Memory	2048 k	Main body of RAM. Consists of four 512k SRAM chips creating a total of 2 megabytes of system memory.
KUSEG	32’h00000000 to 32’h001FFFFF	‘Virtual memory’ - maps to the full addressable 2 M of Main Memory. KUSEG addresses have a ‘000’ prefix. Contains mirrors of KSEG1 and KSEG0.
KSEG0	32’h80000000 to 32’h801FFFFF	‘Virtual memory’ - mirrors KUSEG with caching **enabled**. KSEG2 addresses have a ‘100’ prefix.
KSEG1	32’hA0000000 to 32’hA01FFFFF	‘Virtual memory’ - mirrors KUSEG with caching **disabled**. KSEG1 addresses have a ‘101’ prefix.
VRAM	1024 k	Contains frame buffers, textures, palettes; has a 2k texture cache
Sound RAM	512 k	Contains capture buffers, ADPCM data, reverb workspace
CDROM Controller	1 + 32 k	Includes RAM, ROM, and buffer
Memory Cards	128 k	Extra memory slots that can be accessed through memory-mapped I/O

Figure 6 *- Memory modules with corresponding functions*

To simplify memory accesses on the FPGA, all of the memory units will be lumped together into one large addressable space. This will allow consolidating all interactions between the board and the included DDR3 RAM to one simpler interface. For the VC707, a core is provided for communication with DDR3. To lump all memories into the same physical address space there has to be a translation layer that will take addresses for Main Memory, VRAM, Sound RAM, etc. and convert them into the corresponding physical addresses. The VC707 has a 1GB DDR3 SODIMM card. This is more than enough space for all of the memory in the PSX. Main Memory decoding and caching is handled by the CPU so this means that the memory controller doesn’t have to worry about the conversion from PSX virtual addresses to physical addresses. This means that Main Memory in DDR3 only has to occupy 2048 k. The other memory spaces listed in Figure 6 will be placed offset from each other with different base addresses.

< Insert diagram of memory controller interface with CPU, GPU, SPU, etc. >

Figure 7 *- Multi-device memory controller*

The memory controller has a distinct bus connected to each device that is connected to a memory module in the PSX. Each bus can send read or write request independent of other devices. Since multiple reads or write could be requested at the same time by different devices, the joint memory controller stores requests in a queue with details pointing to who requested the data. Each address sent to the memory controller is offset by a unique amount determined by the device it came from. There is also the matter of varying data widths for each of the busses begin written to 64-bit locations. This is dealt with by using only as many low-order bytes as necessary for each bus. Since the amount of space on the DDR3 card is orders of magnitude larger than what is required for the PSX, it will not be an issue to waste the high-order bits. This allows for a simple and direct translation from PSX address space to the DDR3 RAM.

In addition to the basic memory controller, there are also several DMA channels in the PSX. In the console, occasionally the CPU has to be taken off the main bus to allow other devices direct access to memory. There are five devices that have DMA channels; the GPU, SPU, MDEC, CDROM, and Parallel Port. In total the PSX has seven DMA channels. The GPU and MDEC have two channels each. Each channel has three control registers; the DMA Memory Access Register, the DMA Block Control Register, and the DMA Channel Control Register. Channel selection is handled by the DMA Primary Control Register.

Finally the memory controller will have to keep certain memory-mapped I/O registers that control system signals like polling the controllers exposed outside of the DDR3 RAM so that a read or write doesn't have to be performed to expose those values. These addresses are caught in the memory controller and written to registers to allow fast access and direct wiring to other components.

GPU

The documentation on the original PSX GPU is limited to the outward interface. Thus, our design attempted to implement this interface while including as many internal elements as are know to be part of the original GPU. Figure X contains a basic diagram of how our implementation is laid out.

The GPU in the PSX is limited to rendering only 2D primitives, including lines, rectangles and triangles. The GPU has access to 1MB of VRAM, laid out as a 1024x512x16-bit array. X coordinates range from 0 to 1023, Y from 0 to 511, which each color being 16-bits with a mask bit and 5-bits for each RGB channel (the mask bit used to determine if a pixel is allowed to be overwritten). However, the GPU also allows VRAM to be treated as having 24-bit colors, with RGB channel having 8-bits and no mask bit. This mode is only used by memory transfers (ie, the GPU cannot draw 24-bit color primitives).

The GPU in the PSX receives commands from one of two mapping memory addresses (0x1F801810 and 0x1F801814). Typically, either the CPU writes to these addresses explicitly or sets up a special DMA channel (DMA2) to send a stream of data or commands to the GPU. Commands sent to the first address (0x1F801810) are typically put into a 32-bit, 16-deep FIFO. These commands are any drawing, memory, or drawing parameter instructions; these are denoted as GP0 commands. Any commands sent to the second address (0x1F801814) are not stored on the FIFO and are processed immediately; denoted as GP1 commands. These include instructions such as reset and display mode instructions. Commands consist of 8-bit opcodes and 24-bit arguments. As most commands (expect all GP1) requirement more than the remaining 24-bits for arguments, the following 32-bits in the FIFO contain any additional parameters needs by the instruction; like coordinates and color information for drawing commands. The CPU can also read of these addresses to get status information about the GPU; reading from the first address yields a dynamic parameter or data (set by a special GP1 command); known as the GPU read register. Reading from the second address returns the GPU status register. This register contains a number of current operation status and control bits; including the GPU interrupt bit, video mode bit, drawing enabled bit and more. A complete map is provided as follows:

31	Interlaced Mode
30	DMA Direction
29	
28	Ready to receive DMA block
27	Read to send VRAM data to CPU (via GPU read register)
26	Read to receive command
25	DMA/data request
24	Interrupt request
23	Display enabled
22	Interlaced enabled
21	Color Depth in display area
20	Video Moe (PAL vs NTSC)
19	Veritcal Resolution
18	
17	Horizontal Resolution
16	
15	Textures enabled
14	Flip Textures
13	Reserved
12	Mask enabled
11	Set mask when drawing
10	Allow drawing to display area
9	Dither enabled
8	
7	Texture color mode
6	
5	Semi-transparency mode
4	
3	
2	Texture page
1	
0	

***Figure 8** - GPU Read Register*

The GPU received three types of GP0 commands (all GP1 commands are set flag or reset system and handled immediately as mentioned above); drawing, memory transfer, or set parameter. All of these commands are stored on the FIFO and dequeued by the Decode FSM. This FSM will then execute the instruction by setting various control signals within the GPU. For setting parameter commands, the Decode FSM will immediately process the instruction and change whatever setting is being modified. For example, GP0 opcodes 0xE3 and 0xE4 set the drawing area coordinates; these x-y values being stored in internal GPU registers which as loaded with the new values as soon as the Decode FSM dequeues the instruction. Any command requiring more parameters, like drawing or memory transfer instructions are processed by the Decode FSM by first dequeuing the instruction and saving the opcode in a set aside register. All important parameter information, such as corner vertices for drawing commands and amount of data requested for memory transfer commands, is stored in the Global CMD register (GCMD). Some drawing commands include color information in the first 32-bits (lower 24-bits), so this information is stored immediately. Other parts of the system use the data in the GCMD in order to correctly determine their own operation. For example, the texture unit uses the information in the GCMD to get the correct texture and how to properly blend the text with the primitive's color.

Another type of GP0 command, memory transfers, are more complicated and thus take more power to perform. Memory transfer commands are handled the Decode FSM and a small memory FSM (not shown) to move data from the FIFO to VRAM (in the case of a CPU to VRAM transfer), from VRAM to VRAM or from VRAM to the GPU read register. These commands are used typically to transfer large image data or textures from Main Memory into VRAM.

The third type of GP0 command, drawing instructions, are the most complex and require an entire graphics pipeline to perform. For these commands, the Decode FSM is solely responsible for getting all the needed parameters, including vertices, texture coordinates (inside VRAM) and colors from the FIFO and putting them in the GCMD and initiating the pipeline. The pipeline is a four stage pipeline (excluding the fetch and decode stages which consist of the Decode FSM and FIFO), a Drawing Stage, Color Stage, Shade Stage and Writeback Stage. The pipeline processes 32 pixels, in the form of x-y coordinates in VRAM, in parallel. In order to process a single primitive, the pipeline is filled multiple times by the X-Y Calculator, which continuously feeds 32 new coordinates to the top of the pipeline for rendering. The pipeline does not process all drawing area pixels for each primitive, but blocks the drawing space in 32x32 blocks and determines which of these blocks contains parts of the primitive by simple minimum x-y, maximum x-y comparison.

The GPU can render 3 basic primitives; triangles, lines and rectangles. Polygons are either 3- or 4-vertex shapes, and can be colored, textured and shaded. 4-vertice polygons are rendered as two 3-vertices polygons inside the GPU, with the first 3 argument coordinates forming the first and last 3 coordinate arguments forming the second. Lines are defined by 2 vertices and can be colored and shaded. Some commands in the GPU specify “poly-lines” which are simply a multi-point stream of vertices, with each two forming a new line. Poly-lines are like 4-side polygons in that the first 2 vertices are rendered as a single line, then the next vertex from the FIFO is used to for a line with the second coordinate from the previous segment. A special en code (0x55555555) is sent to denote the end of the poly-line. The third type of primitive, the rectangle, is simply defined by a coordinate for the top-left corner and a width-height pair. Rectangles can be colored and textured by not shaded. All primitives can be turn semi-transparent; or mixed with the pixel currently at the point.

In order to draw and color these primitives, 32 pixels are send through the drawing pipeline. In the first stage, the given 32 pixels are determined to be inside or outside the primitive. For rectangles, a simple comparison is done. Both triangles and line rely on a special “line finder” module to determine what points are contained in them. The line finder works by determined whether a point is on, “above” or “below” a line defined by 2 points (The module uses some techniques similar to barycentric coordinates; taking a determinant, so it uses two multipliers, several adders and a bit of combinational logic). Using this module, the lines can clearly be draw (by snipping the lines based on the maximum/minimum x-y from the segment). Triangles can also be drawn using this module. Three of these modules are used near the GCMD to determine which side of the line formed by 2 of the triangles vertices the third point is on. By doing this for all three vertices, a set of three sides is used by the line finders in the drawing stage to check if the processed pixel is also on the same side of the three lines as the third point. If this is true for all the vertices of the triangle, the point is within it. A map for which points are contained within the primitive (in the form of an in_shape bit for each pixel) is passed along the pipeline.

The next two stages, the Coloring and Shading stages are not yet implemented. However, some detail about what they do is provided in the documentation. The color stage is responsible for applying the single color or texture to the primitive. This consists of a texture cache and a Color Look-up Table (CLUT) cache. As image data can take up a lot of space, the PSX allows textures to be stored in 3 color modes, 15-bit full color, 8-bit and 4-bit compressed modes. The 8-bit and 4-bit modes store indices into CLUTs instead of actual color data. These CLUTs are stored as 256x1 or 16x1 images in VRAM. To save memory accesses, the GPU caches texture and CLUT data in small caches. The texture is mapped to the primitive via texel coordinates provided by arguments to the command (this texel coordinates are stored in the GCMD). While polygons may have scaled textures, rectangles may not have scaled or rotated textures (note that the texture flip bits in the GPU status register do allow for simple texture manipulation for rectangles). Shading can either be Gauraud or Flat; but providing colors for each vertex as arguments to the command. Whether a shape is shaded or textured is part of the opcode.

The final stage, the Writeback stage, writes any pixels who are inside the shape back to VRAM. However, it first reads in the pixel values at that coordinate in VRAM to check the mask bit and perform transparency blending if needed. If the mask bit for a pixel is set, the new pixel will not be written back over the current one.

This pipeline allows all the PSX primitives to be draw. VRAM is dual-ported to allow the video display module (HDMI) to read the drawing data and display it on the screen.

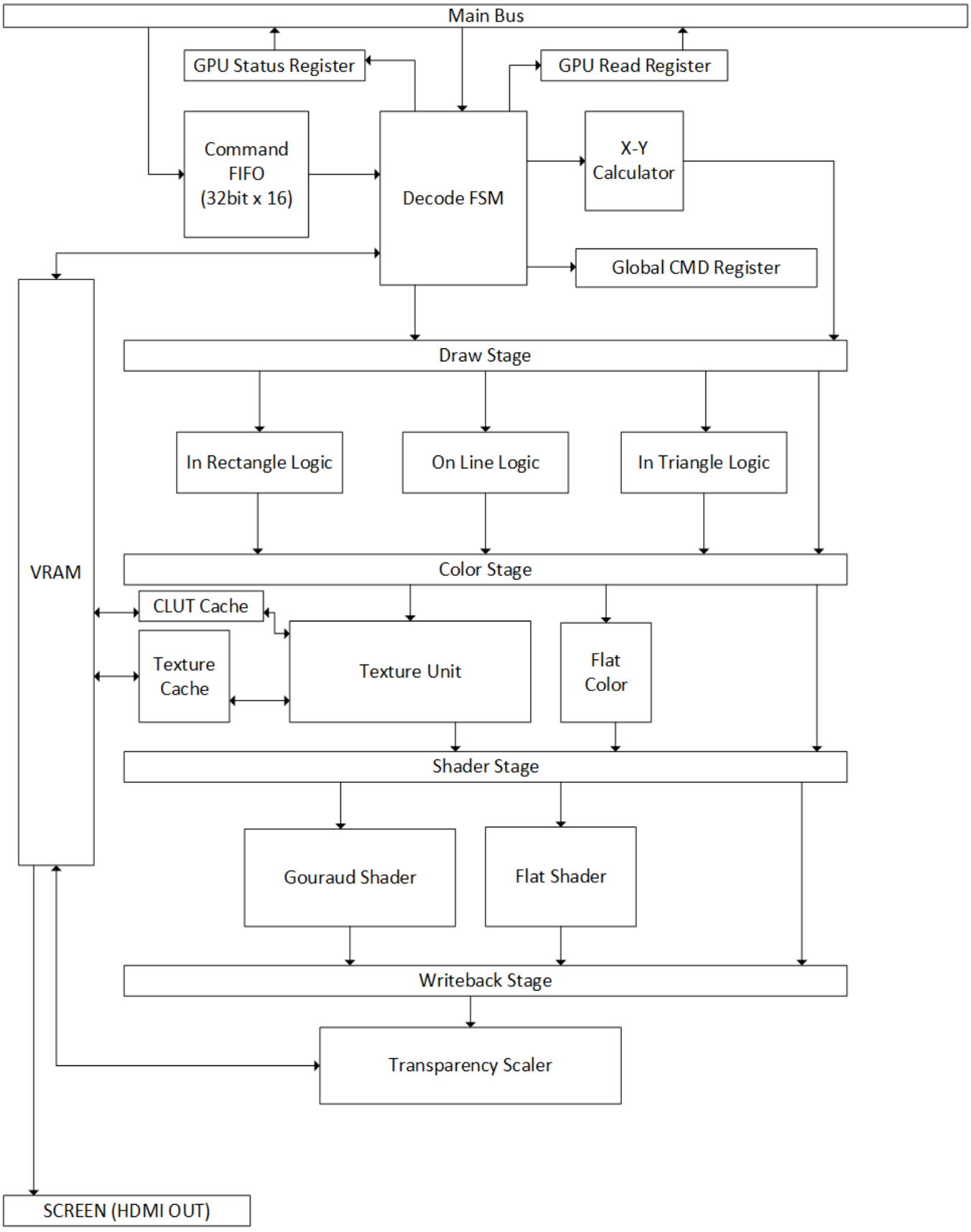

Figure 9 *– GPU block diagram*

MDEC

The MDEC (Macroblock Decoder) unit is responsible for decoding compressed image data from the CDROM and transforming it into full 24-bit color representations to be stored in Main Memory. The MDEC has access to two DMA channels for retrieving and storing compressed and decompressed image data in Main Memory. Compression of images and video frames allows PSX games to have more content in the limited memory space on CDROMs.

The compression scheme used by the PSX is similar to JPEG file format. The decompression algorithms are provided in the documentation, though we haven't had a chance to look over them yet.

SPU

The SPU handles all sound capabilities on the PSX. It handles 24 voices (channels) and has a 512KB sound buffer, along with various filters. Data is stored in compressed blocks of 16 bytes, each with 14 packed sample bytes and two header bytes, one for the packaging and one for sampling and looping. The memory layout is small and simple: CD audio L and R, voice 1 and 3, system, sound area, and reverb area. It has few commands for pitch and reverb operations. It has several 16-bit registers which handle control and data information and status. There is a special effects processor in the SPU which does reverb and echo delay type effects, but it only does one effect at a time. There is quite a bit documentation on the overview of the SPU, but details have not been heavily searched for just yet as this component is implemented later in our schedule.

CONTROLLER

The standard digital controller of the PSX features 14 buttons, active-low, and uses 8-bit serial communication to talk to the game system. There are also other controller formats that include analog or digital joysticks (See Figure X). These controller still talk to the system over the same serial communication protocol and have additional bytes they have to transfer.

Figure 10 *- PSX Dualshock Controller*

The serial interface between the controller and the PSX follows a relatively standard master-slave configuration for serial communication. The PSX is the master. It is responsible for initiating all data transfers and is also in control of the clock. The controller has no input to the clock signal and cannot force the system to hang (unlike the I2C serial protocol which is discussed in HDMI). There are nine pins connecting the controller to the PSX (See Figure X). Of these pins, five of them are critical to data transfer; DATA, COMMAND, ACK, ATT, and CLK. There are two pairs of analogous pins. DATA and COMMAND carry the serial information. DATA is sent from the controller to the PSX, and COMMAND is sent from the PSX to the controller. Similarly ACK and ATT are signals from the controller and from the PSX respectively. ACK tells the PSX when a byte has been transferred. The absence of an ACK indicates the completion of a data transfer. ATT is pulled low by the system when it wants to poll a controller for information. CLK is the the clock signal that drives the serial communication. CLK is generated by the PSX. The PSX console uses a 250kHz clock for communication with the controllers. However it should be noted that the controllers are able to operate correctly with a clock frequency anywhere between 100kHz and 500kHz. The controller operates at a range between 3.3V and 5.0V. Because the VC707 only supports

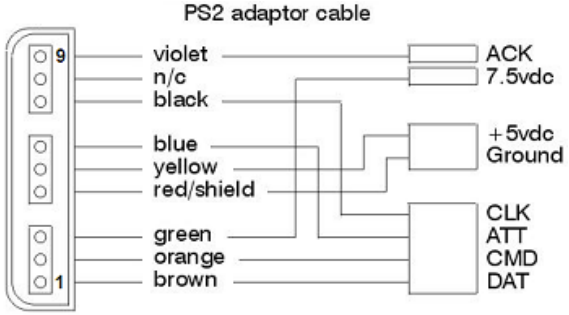

Figure 11 *- PSX Controller pinout*

output voltages of 1.8V it is necessary to use a voltage translation circuit to communicate with the controller. Another point worth noting is that one can get away with using only four pins per controller. The ACK signal is not entirely necessary on the receiving end as long as the transmission is begin carried out correctly. The only true purpose of ACK is to signal when a transmission is over in this particular protocol, however since the PSX controllers will either always send five bytes if digital or 9 bytes if analog, it isn't necessary to have the ACK signal. Errors in transmission can be caught and handled without the need of ACK.

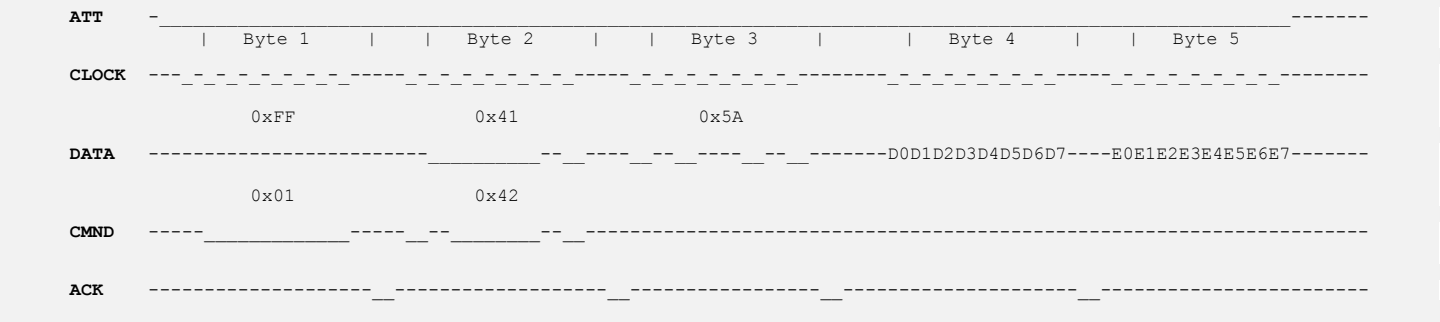

Figure 12 *- Controller Timing Diagram*

Data transmission with the controller is initiated when TXEN (from register JOY_CTRL) and register JOY_STAT.2 are set high. This indicates that the system is ready to poll the controller for its values. The first action is the system pulling ATT low. ATT is held low for the duration of the transmission. In the first byte, the PSX sends 0x01 (start command) to the controller asking it for an identification code. This code lets the PSX know what type of controller is being used (Digital, Analog, NegCon, etc.). The controller doesn't send anything back on the first transmission. In the second byte, the controller responds with its type identification and the PSX sends 0x42 to request data. The third byte marks the end of the handshake between PSX and controller. The controller sends 0x5A indicating that it will now start sending data. The rest of the transmission is all data, and the number of bytes sent depends on the type of controller. A digital controller sends a total two bytes of data. In comparison an analog controller sends six bytes. Between each byte the controller pulls ACK low, but as mentioned earlier, it isn't entirely necessary to wait for the ACK. On the final byte, the controller does not pull ACK low, thus indicating the end of transmission and the PSX responds by releasing ATT. Signals from the controller are active low.

ROMS/GAMES

Normal operation of the PSX dictates that games are played from CDROMS. The console has CDROM reader and the appropriate controller interface to read from the CD. The CDROM controller has a few dedicated addresses in the memory-mapped I/O that give the CPU a channel to send commands, to send parameters for the commands, and to read data into a buffer. The CDROM controller in the PSX is also capable of playing audio CDs, but this functionality isn't a priority for this project.

Interfacing with an actual CDROM controller has several challenges and should be avoided if possible. There is a completed project called PSIO which is basically a piece of hardware that interprets the CDROM commands sent from the CPU, translates the commands into actions to read from a ROM loaded into an SD card or other form of flash memory, and then sends data back to the CPU in the desired output format. This card is plugged into the Parallel Port of the PSX. If it is possible to get a hold of the HDL for this CDROM emulation hardware, then the SD card reader on the VC707 can be used to load and play ROMS. There is still more to explore and learn about this. Hopefully it will be possible to get either the hardware or the HDL for the PSIO project.

HDMI

For standard outputs of audio and video the Virtex 7 (vc707) board only has HDMI. An intermediate chip, the ADV7511, made by Analog Devices facilitates (lol, facilitates….) communication between the FPGA and the HDMI output. It also handles many functions specific to the HDMI spec including but not limited to; Data Encryption, EDID processing, etc.

The ADV7511 chip has several configuration registers that have to be configured before it can operate properly. Configuration of these registers is handled over the I2C bus. Initially we created out own I2C interface using a previous team's code as a model. There is also a reference design available from Analog Devices which we used to find/verify which registers needed to be configured. However this wasn't leading to much success. There was still confusion regarding how to deal with inout ports in Verilog. However by this point we already transitioned to using an IP Core from OpenCores. The transition to the IP Core didn't solve the issues we were facing, but it did clarify the proper use of inout ports. Careful testing with a drastically reduced clock showed the correct output from the FPGA, but not acknowledgement from the ADV7511. Later we discovered that there is a MUX on the I2C line of the VC707 that has to be configured as well before anything can be sent to the ADV7511. This mux sits between the FPGA and eight different slave devices. Once configuration of this MUX was sorted out, and all the inout ports correctly wired, finally ACK signals were begin received from the ADV7511. To verify that the data being written over I2C, out interface was modified to perform a read following each and every write to ensure that the configuration was being done exactly as we wanted it. This verification led to further confusions. Of the 62 registers written to for configuration, only one failed to read back the same value as its write. This is the register responsible for setting the ADV7511 in HDMI or DVI mode (this basically means audio ON or OFF). Despite setting the register to be in HDMI mode, it would always read back in DVI mode. This led to a curious observation. When first testing out HDMI and using the reference design to experiment, we could not get sound to play no matter what we did even though the reference design claimed to play 'clicks' of audio. It isn't just an empty claim either, the code clearly performs a simulated audio DMA and it also configures the registers to play audio as well as video. Currently we are trying to work with Analog Devices to troubleshoot this anomaly.

The audio for the ADV7511 can have several protocols (selected by the control registers). However, on the VC707 board, only the SPDIF protocol seems to be available. As we could not find any good ip cores for SPDIF (Core Gen has a non-synthesizable one, and OpenCores has a VHDL one which had other problems), we decided to create our own. The SPDIF protocol is pretty straightforward and did not take too long to implement. We simulated the module with some sample data (gotten via a simple FSM reading from what will be an on-chip ROM) to ensure the waveform for SPDIF was correct. After some debugging, we were able to get the waveform to match the protocol standard. Unfortunately, since configuration has not been complete, we do not yet know if the SPDIF module will work with the ADV7511 chip.

The video for the ADV7511 appears to follow the timing protocols of VGA. It takes in the familiar VGA signals of HSYNC, VSYNC, and DE, as well as an HDMI/ pixel clock. The documentation was a vague about whether it also followed the front porch/ back porch protocol of VGA and its example chart values used terminology different from those commonly used to describe VGA. At any rate, the video module currently outputs the waveforms corresponding to a VGA 720x480 video frame. It uses a simple two state FSM to go from frame setup to data enable, and most of the sync output is handled by counters that are managing the horizontal and vertical resolutions. For testing purposes we are sending one hard-coded color to the chip. We have not been able to see the video output due to a hold on the configuration.

Where We Are Now

Progress is good for everything except for HDMI. Looking at the Gantt chart (Figure 1) the only task that was pushed back was the HDMI configuration. This led to a slight setback in work on the controller and memory, but that should be rectified within the next week. HDMI issues are also preventing us from testing video and audio, so those are two other areas that could require an unpredictable amount of time for debugging purposes. At this point we are working hard to keep on top of all the other components so that if and when we get HDMI sorted out, we can take the time to return back to audio and video.

Also, once we the memory module is completed, we will be able to load the BIOS onto RAM and then test the CPU and GTE on the FPGA. Memory probably the next most significant roadblock to continuing testing and maintaining forward progress.

On the user input side, the controller interface is functional and ready to be integrated into the system. There is some additional work that needs to be done to hook up the controller to memory-mapped I/O, but that should be relatively straightforward once the memory controller is completed.