Team PSX

Anita Zhang

Arnob Mallick

Mike Rosen

Carnegie Mellon University

Electrical & Computer Engineering

18-545

Fall 2013

“Satisfaction lies in the effort, not in the attainment.

Full effort is full victory.”

∼ Mahatma Gandhi

Contents

1 Overview

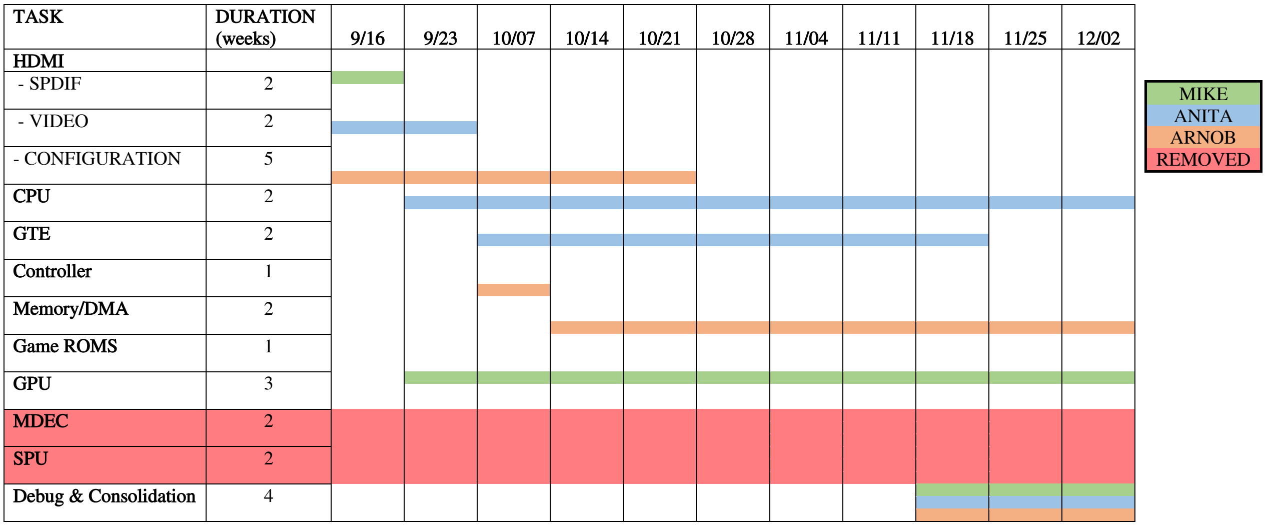
Our team's original goal was to recreate the original Sony PlayStation (PSX) on the Virtex-7 FPGA VC707 Evaluation Board. We planned to implement all major components including sound, video, controller input, and game data from ROMs. Although our goals remained constant throughout the semester, a few key alterations were made to the original goal. Over a month into the semester we made the decision to switch to Altera's DE2-115 Evaluation Board and dropped some major components including sound. The reasons for these changes will be discussed later in the report.

The PSX was originally planned as a new version of the SNES with CD-ROM input rather than the traditional cartridge games. However, as Sony continued to work on the project, Nintendo, the makers of the SNES, grew increasingly frustrated with the development of 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 competitor's chip, Sony decided to continue the project alone and create their own gaming console for the market. A few years later, in December 1994, the PlayStation was released in Japan. The console was released in the USA the next year in September 1995. The PSX contained the following components:

- MIPS I, 32-bit CPU with 2 co-processors and 2MB of RAM
- 2D GPU with 1MB of VRAM
- SPU capable of generating 24 voices with 512KB of RAM
- CD-ROM reader for reading game discs
- New controllers with 10 buttons (two joysticks added to the controllers in 1997)
- 2 slots for EEPROM memory cards

We attempted to reconstruct these components as close to the original as possible. However, due to the period in which this console was made, availability of documents is sparse and the quality of documentation is often questionable. This means that most of the components we have recreated are not identical to their counterparts in the original system. Video output on the Altera board is handled via VGA while the PSX used Composite Video. While the GPU in the PSX was responsible for handing video RGB data output to a video digital to analog converter (VDAC), our system's VGA module directly reads VRAM data sharing the VRAM bus with the GPU. Also the omission of components such as the MDEC and SPU requires providing a dummy interface that will behave in a predictable manner and provide the CPU with signals that emulate "normal" operation. There are several other alterations that will be explored in the later sections. In the following sections, we will work our way through all the components of the PlayStation describing the original hardware alongside our implementation and important details we learned in the process.

1.0.1 Schedule



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	5	🟧	🟧	🟧	🟧	🟧						
CPU	2		🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
GTE	2			🟦	🟦	🟦	🟦	🟦	🟦	🟦		
Controller	1			🟧								
Memory/DMA	2				🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
Game ROMS	1											
GPU	3		🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
MDEC	2	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
SPU	2	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
Debug & Consolidation	4									🟩 🟦 🟧	🟩 🟦 🟧	🟩 🟦 🟧

🟩	MIKE
🟦	ANITA
🟧	ARNOB
🟥	REMOVED

The further we got in the semester, the more behind we got in our project. It was a mutual understanding, but had to shave components in order to meet our goals.

2 Tools & Platform

2.1 Xilinx

2.1.1 Virtex-7 FPGA VC707 Evaluation Board

We started off the semester using the VC707. The VC707 looked like a promising option for us because of the huge number of resources it carried and its sheer power. We believed that since we were going to try to implement a PlayStation, we would be needing all the FPGA logic units we could get. The Virtex-7 bit us in the back with two critical obstacles that we were not able to fully overcome. The first was a combination of the I2C Bus and the ADV7511 HDMI chip. Second, the board only has eight GPIO pins.

2.1.2 Vivado Design Suite

Vivado is nice. It runs on Windows (even Windows 8 64-bit). It also supports SystemVerilog. The ChipScope tool packaged with Vivado is easy to configure and is an extremely valuable debugging tool. But it is slow, even for the simplest of designs.

2.2 Altera

2.2.1 DE2-115

Roughly half of the way through the semester we switched to the Altera board, hoping to remedy the roadblocks we faced using the VC707. With the new board we had all the GPIO pins we could ever need. Also we were able to switch to a much more familiar VGA interface for video output. The less powerful DE2-115 came with a few drawbacks. On-board memory was limited and we had to come up with some creative solutions to work with what was available. Also the Altera board is much slower than the Virtex-7, so we ran into timing constraints that were not issues before.

2.2.2 Quartus

Its synthesis mechanism is dumb and leaves something to be desired. At one point we managed to get it to use less than 1% of logic units, which is completely wrong. But when simulation starts taking hours to reach the same point the actual hardware can reach in 10 seconds, you will need a new way to look at waveforms. For Quartus, that is SignalTap II. It may take some experimentation to find the limit of how many signals can be viewed at once. Asynchronous resets are a must. Being able to change the trigger point such that signals before and/or after the trigger can be viewed is very useful.

2.3 Miscellaneous

2.3.1 No$PSX Debug Emulator

It was amazing. It was like Christ's resurrection or the Buddha's enlightenment. Finding this tool was a religious experience unlike anything before it. This messiah of software was discovered right in time to guide us through the desolate path of integration and debugging. A true miracle. Giving pause only to silently crash frequently, No$PSX proved itself of being worthy of god-like praise and admiration. With its every touch, the clouds of evil and hatred departed and the way towards a successful BIOS boot laid bare. If only we could follow this magnificent beast into the shining gold yonder. Alas it was not meant to be.

2.3.2 Synopsys VCS

Our go-to simulator from 18-341 proved to be very useful in debugging our modules. The GPU and CPU, especially, benefit from using VCS to run simulations and view waveforms of various signals.

2.3.3 SPIM MIPS Simulator

The best way to test a MIPS. Write assembly, run it through SPIM and you will have perfect hex codes for your Verilog CPU. Use system tasks to feed the codes to memory and youre set for CPU testing. One idea that came up late in the game is to test CPU, memory, and GPU integration by using the CPU to load the right instructions into memory and have the GPU execute them. Writing Pong in MIPS is one way to do this (see "What Needs to be Done").

2.3.4 Python

Python was an invaluable resource for converting files and quickly scripting useful things. For GPU debugging, python served to convert easily readable files of instructions into HEX files for both simulation and synthesis as well as convert VRAM HEX dumps from the GPU testbench into PNG so that VRAM could easily be viewed and verified.

3 Components

3.1 System

The PSX can be broken down into 9 primary parts. These are the CPU, GTE, GPU, MDEC, SPU, Controller Interface, Main Memory + DMA, ROMs/Game Data, Audio-Video Interface. As seen in the schedule, we planned out times and members to work on each piece.

Figure 3.1: Original PSX

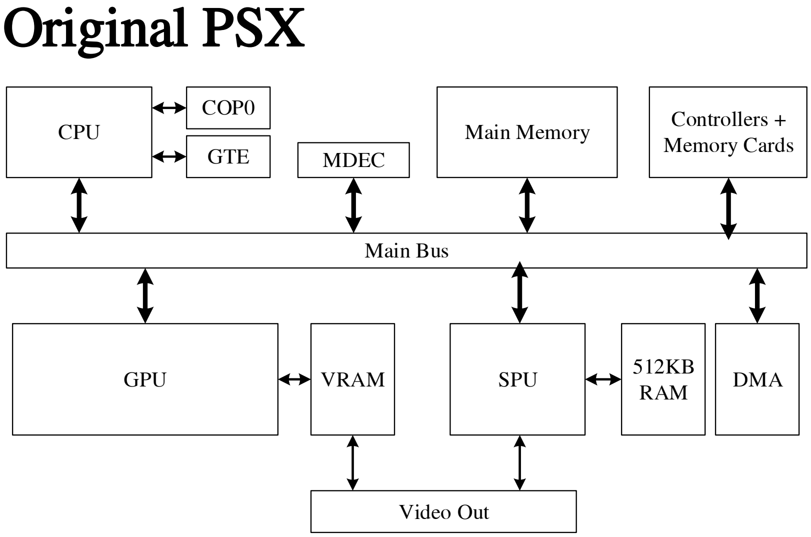
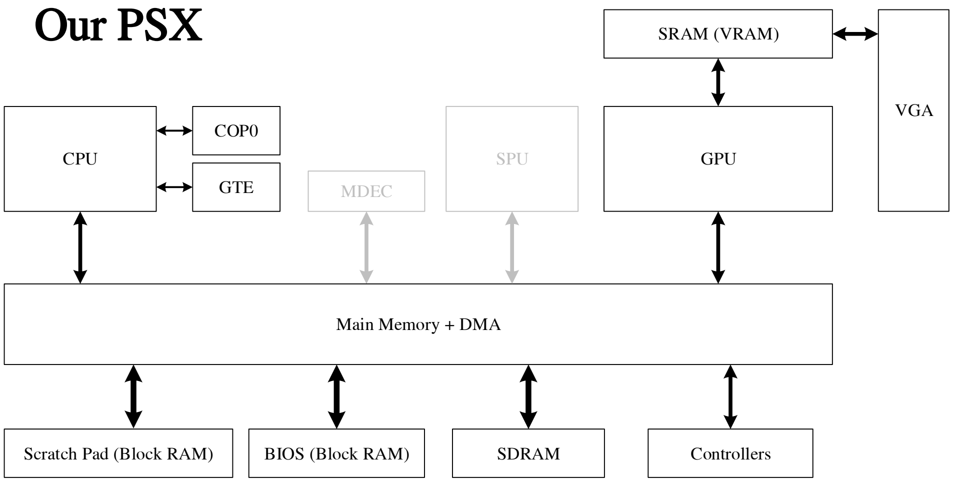


Figure 3.2: Our PSX



3.2 CPU

3.2.1 MIPS

The CPU we are using is an OpenCores MIPS CPU written by the University of Utah for the XUM project/architecture. The schematic pictured at the end of this section is the one they created. 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 exceptions and interrupts in the CPU.

A classic RISC pipeline contains five pipeline stages for instruction fetching, decoding, execution, memory access, and write-back. The key points where each implementation of the MIPS ISA differs comes from branch execution and data/control dependencies. In the OpenCores MIPS they have chosen to implement a single branch delay slot (standard MIPS feature) instead of stalling. For performance reasons they have also implemented data forwarding. The data forwarding logic is based on a want or need condition contained in each instruction when it gets decoded. Each stage in the pipeline then uses this to check for data availability. For example, if data is available in the memory stage and the decode stage wants/needs it, then the data will get forwarded. However if data is still not available by the time the instruction reaches execute, and it needs it to perform an ALU operation, then the execute stage will stall until the data is available. The hazard detection logic and forwarding logic live in the same module.

The memory interface in the OpenCores MIPS is very robust. When the CPU has a memory request, it sends the request high. It then waits for the memory controller to send the acknowledge high, the earliest being on the next cycle. The CPU will then go low the cycle after, and the handshake is complete when the memory controller goes low. This handshake worked well with our memory controller because the controller had to service more than just CPU requests.

It is important to note that the OpenCores MIPS does not implement virtual memory operations, nor cache operations. The original PSX implements virtual memory the way the MIPS standard does it, 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. The OpenCores MIPS did not handle this, so our translation happens in the memory controller.

Coprocessor 0 (Cop0) is what handles virtual memory and exceptions in the MIPS processor. It contains 32 registers for exceptions, CPU parameters, and various debug and cache options. Exceptions can be divided into two categories: synchronous and asynchronous. Software interrupts and 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. Allowing forward instructions to run to completion is easier on for the hardware so it consistently takes that approach. 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.

As mentioned before, the OpenCores MIPS does not implement virtual memory, and thus does not have a TLB implemented. Any form of caching available is also unavailable. We have come up with some hacks around this, mentioned below. 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, we are disregarding. The LR33x30 datasheet that exists does not contain anything deviating greatly from a standard MIPS.

3.2.2 GTE

The Geometry Transformation Engine (GTE) is the heart of all 3D calculations in the PSX. Anything involving vector/matrix operations, perspective transformations, color equations, etc. are done by the GTE. It has about 20 instructions, with parameters encoded in the instructions, especially suited for this task and thus is much faster than a general purpose CPU for doing 3D calculations. It also contains several special load/store instructions for accessing the 32 32-bit control registers and 32 32-bit data registers. Loads/stores through the CPU are documented to have a delay of at least 2 cycles. And there cannot be multiple GTE instructions running at once.

The GTE is mounted to the PSX's MIPS as coprocessor 2 (Cop2). To program the GTE, the appropriate bits in the status register of Cop0 must be set in order to enable Cop2. What this meant for us is that the relevant portions of Cop0 disabling Cop2 were removed. One of the great features of MIPS instruction encoding is that the first 6 bits of the instruction indicate which coprocessor it is issuing commands to, or which kind of instruction it is executing. 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.

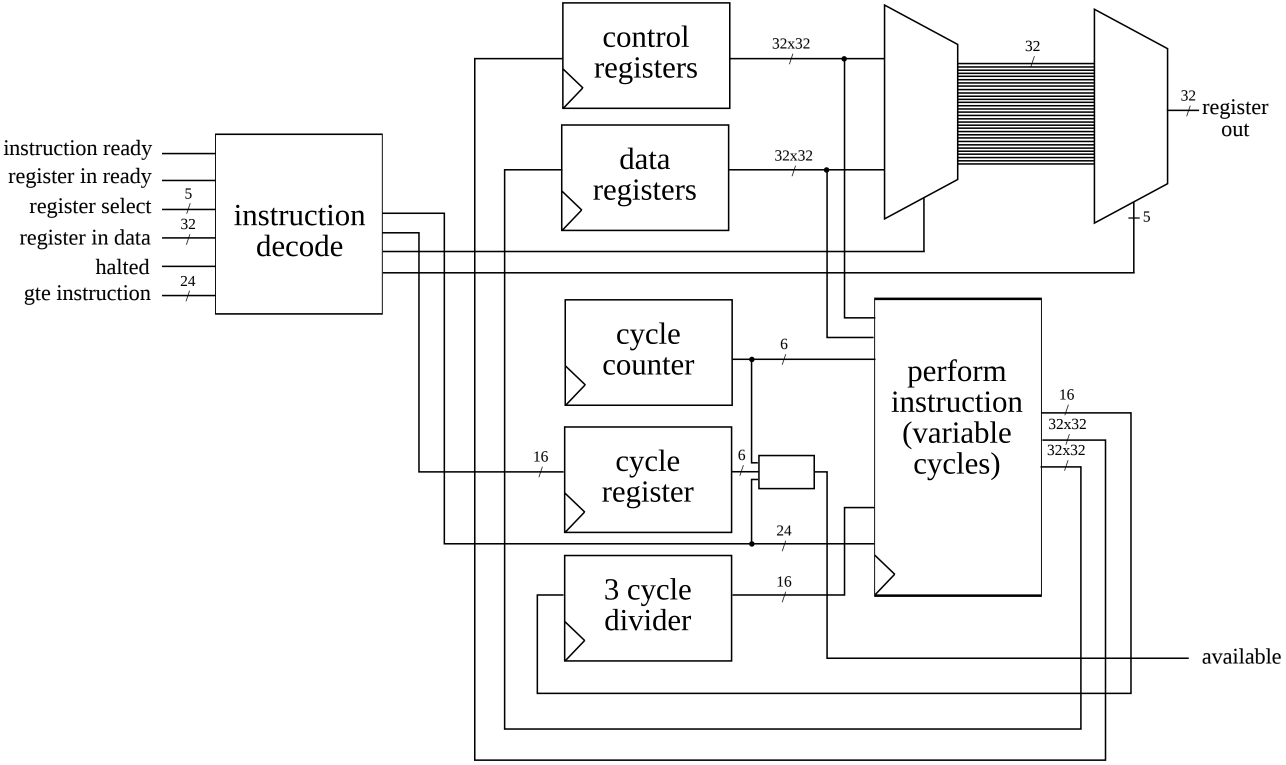
The control registers in the GTE contain all of the scale factors, offsets, rotation elements, etc needed for the various transformations. Although each register is 32-bits wide, most of them contain two 16-bit registers in specified signed fixed point format (most of them are 1.3.12). The data registers 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 below).

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. The order of multiplication matters.

Implementation was made easy with a fantastic document detailing each instruction (reduced to arithmetic operations) was found. It contains the instruction, cycle count, equations needed to perform the instruction, and which registers are read and modified during the operation. This shows us how the PSX GTE works at a high level and enables us to create a cycle accurate implementation of the GTE, with accurate modifications of registers. Due the resource constraints the GTE right now is not timing compliant, but it operates correctly.

For our project implementation, the GTE sits around the decode stage of the CPU with Cop0. When a GTE instruction is decoded, a flag is set telling the GTE which instruction is entering and the GTE buffers the necessary information to run the instruction. This includes the parameters passed into the GTE, the command itself, and the number of cycles this instruction will take. The GTE then sets a flag telling the CPU that is is unavailable for more GTE instructions. Each instruction is individually pipelined; that means that depending on what cycle it is since the instruction started, it will execute a different part of that instruction. Because each instruction was specific and has a different number of cycles, there was no obvious way to pipeline it according to operations. On the last cycle of an instruction, the counter and all internal signals used for calculation are cleared. It takes at least 2 extra cycles more than what is specified in the GTE document in order to latch the command, and clear the signals before the next command can come in. Two commands use division, but a single cycle divider does not meet the timing requirements of our system on the Altera boards. This was replaced with a 3 cycle pipelined divider from the Quartus IP cores library. A high-level datapath can be viewed below.

Figure 3.3: GTE Datapath



As stated before, the PSX is documented to take at least 2 cycles to access the registers in the GTE. In our project, it may take fewer than 2 cycles depending on the operation. During CPU and GTE integration, it was easier to have the register to register data for reads or writes available by the next cycle so as to combine it with Cop0's logic in the CPU. This made it easier to implement CFC2, CTC2, MFC2, and MTC2 (CPU/GTE register to register moves) the same way MFC0 and MTC0 (CPU/Cop0 register to regsiter moves) were implemented. Forwarding from any stage made this possible.

Memory to register operations to and from GTE to CPU to memory were a little trickier. Writing from a GTE register to memory was easy; the GTE data was available immediately so it could easily propagate down the CPU pipeline with a GTE flag telling the memory stage to write GTE data instead of other data. Reading from memory and writing to a GTE register was trickier. To avoid any potential forwarding that might occur and pollute the pipeline with incorrect data (ie. the CPU sees a write to register 14 of the CPU but really it is writing to register 14 of the GTE), these memory reads stall the whole pipeline until data is written back to the GTE. A flag on the CPU's register file enable prevents write back to the register file.

Stalling for multiple GTE commands was trivial; it checks the available bit in the GTE and the Cop2 bits in the decoded instruction to see if it needs to stall. Stalling the pipeline at the right time for memory operations was the most difficult part of this integration.

One of the challenges of implementing the GTE was the fixed point math and overflow handling. The command list does say which flags to modify in case of 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. The way it is implemented now is to truncate the necessary bits according to the format. Every operation stores in a much larger register than needed so as to set overflow bits correctly. There was no good way to test edge cases, and so the GTE is working for basic cases only.

3.2.3 CPU Modifications

Because this is a standard MIPS, the OpenCores version probably assumed users would look at the ISA rather than want implementation specific information. Because of this, the

OpenCores version is not entirely documented. Fortunately, the Verilog has a consistent style that had odd variable names, but was easy to read. Comments were added for major design decisions (exceptions, memory operations, etc), but it took a series of redirects to find everything.

To test the CPU, the Spring 2013 18-447 MIPS testbench was integrated and run with a batch verify operation on all the MIPS programs available from the course. This covered most of the standard operations, which includes jumps/branches, ALU operations, and multiplication and division instructions. A lot of tests needed modifications because the 18-447 MIPS was not designed with a branch delay slot.

The OpenCores MIPS was written for big endian memory accesses; the PSX ran with little endian. To force it into little endian mode a define statement in the memory module forces the CPU into little endian ragardless of what bits are set. On that note, reading the BIOS in 32-bit blocks is not correct; read 32-bit blocks, reverse the endianness of each block, and then feed them into the CPU.

During these runs of batch verification, it was discovered that the OpenCores MIPS implemented MTLO incorrectly. Since the LO and HI operations of the CPU lived in the ALU, this meant the ALU took in two operands as inputs. Instead of moving in the first operand to the LO register during MTLO, OpenCores MIPS was moving in the second. Simple fix, but it shows the importance of verification.

During BIOS runs, we found that Cop0 was not behaving correctly during an event that wrote to the exception program counter (EPC). The first exception wrote successfully to EPC, but later exceptions could not write to EPC. It turns out that when an exception occurs, MIPS specifies that it will set the exception level (EXL) bit in the Status register of Cop0 as well as write EPC. The EXL bit does a few things, but the relevant thing it does with EPC is disable writes to EPC during later exceptions. When running on the No$PSX debugger, EXL never gets set. One of the undocumented subtleties of the PSX CPU.

The PSX does not explicitly use any of the standard MIPS interrupt ports. Instead our implementation creates a new input to the CPU that will take the 10 hardware registers needed by the PSX. These values are then latched in the top module of the CPU so as to avoid having to read from memory to see if an interrupt has been acknowledged. Both the interrupt mask (I_MASK) and interrupt status (I_STAT), are hardware registers which live at address 0x1F801070 and ox1F801074 respectively are latched. Reads to these locations return the values in the CPU. Writes go to both the CPU latched version and to memory. To set up execution of a hardware interrupt, I_MASK and I_STAT must be non-zero; this sets the Cause register (r13) bit 10 in Cop0 to high, mimicking a standard hardware interrupt being received. If the Cop0 Status register (r12) bits 0 and 12 are also set, indicating interrupt enabled and interrupt mask, respectively, then the interrupt gets executed via normal hardware interrupt facilities. This means it jumps to the exception vector and executes until a return from exception (RFE) instruction.

One trip up during BIOS testing was cache initialization. Because we had no cache facilities implemented, all memory writes wrote out, regardless of whether it was meant for the cache or not. One of the things cache initialization does is set a PSX defined bit called Isolate Cache (Cop0 Status register bit 16) which means all access will now be cache only. To ignore these memory operations, the CPU checks for that bit and during decode it will override the impending memory access, turning it into a NOP. This got us past the cache initialization stage of the BIOS, because it prevented overwriting memory locations at the wrong time. But this may be an issue later on if those values need to be read from at a later time.

3.3 GPU

3.3.1 Overview

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. All opcodes and their associated instruction are provided in Appendix A.

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 below (GPU Status Register table).

Table 3.1: GPU Status Register

Bit	**Description**
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 Mode (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	

The GPU has three types of GP0 commands (all GP1 commands setting flags or reset the system and are 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 several small memory FSMs 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 1 pixel, in the form of x-y coordinates in VRAM, in parallel (limited to 1 due to board constraints, but the pipeline width can be parametrized). In order to process a single primitive, the pipeline is filled multiple times by the X-Y generator, which continuously feeds 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-vertex 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.

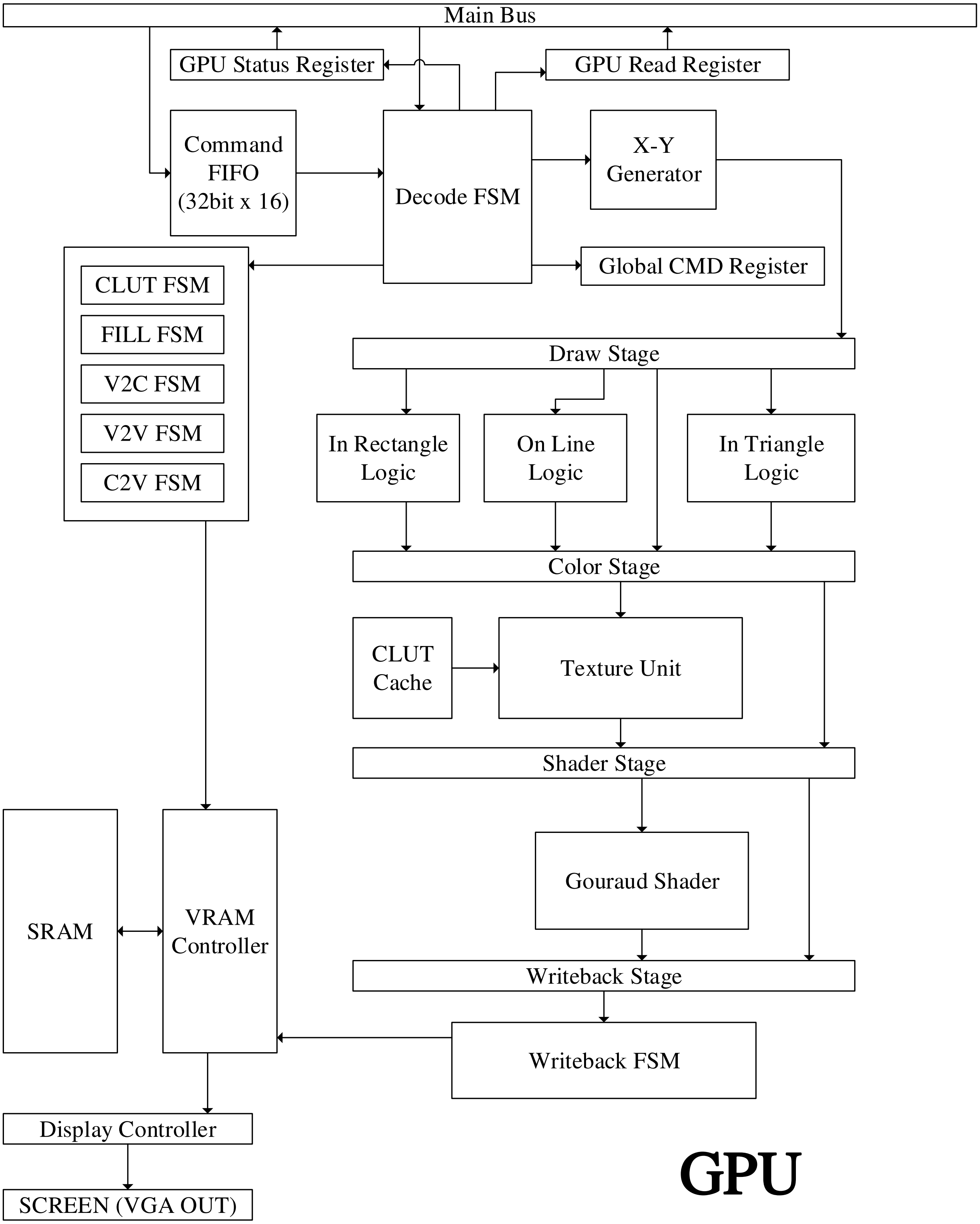
In order to draw and color these primitives, 32 pixels are send through the drawing pipeline. In the first stage, the given 32 pixels are determined to be inside or outside the primitive. For rectangles, a simple comparison is done. Both triangles and line rely on a special "line finder" module to determine 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 stage is the color stage, which is responsible for applying textures to the primitive (or leaving it untextured if it is not a textured shape). The color stage is split between calculating the texture coordinate within the texture page (the location in VRAM where the texture data is stored; either given in the command or in a special register in the GPU set by GP0 commands) and applying the texture to the primitive. As image data can take up a significant amount of memory, textures in the PSX can take be either 4-bit, 8-bit or 16-bit color. For 4-bit or 8-bit, each pixel represents an index into a color look-up table (CLUT) which exists as a 16x1 or 256x1 image in VRAM. If the primitive being decoded is textured, the decode FSM will initiate an FSM to retrieve the CLUT from VRAM and store it in a cache. Just as in the original PSX, the GPU has a 256x16 CLUT buffer able to store a single CLUT for the primitive currently being drawn. However, unlike the original PSX, which had a small texture cache for storing the texture data, the GPU in our implementation retrieves the data from VRAM for each interpolated texture coordinate and stores it in a texture buffer. provided by arguments to the command (this texel coordinates are stored in the GCMD). While polygons may have scaled textures (as a result of the interpolation), rectangles may not have scaled or rotated textures (note that they may be flipped along the x or y axis based on GP0 command 0xE1). This process uses the writeback FSM and triggers a pipeline stall until the texture buffer is full. The texture buffer stores the 4-bit, 8-bit or 16-bit color which is translated by using the CLUT if needed in the second color stage. The first 15 bits of the texture is used for color information while the 15th bit is used to determine the transparency of the pixel. This setting overrides the semi-transparency flag in the status register and is applied in the writeback stage.

The shader stage applies the final color to the primitive (before blending). This means either using the color provided in the GCMD or uses the interpolated colors for gouraud shading. As this calculation is also complicated, it is split into two pipeline stages. These are done in 32-bits then saturated to the 24-bit color used by the writeback stage. The calculations are further explained in the interpolation section. 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 drawn. VRAM is dual-ported to allow the video display module to read the drawing data and display it on the screen.

Figure 3.4: GPU Pipeline



3.3.2 Interpolation

Linear interpolation is the technique used by the PSX to determine intermediate colors when shaping or applying textures to polygons. The technique is essentially a remapping of x-y coordinates into the color or texture plane. That is, the given x-y coordinate inside the shape is linear transformed into a color or texture coordinate using the following equation:

$$c_x x + c_y y + c_s = n$$

Where x and y are the x-y coordinates, n is the interpolated coordinate, and c_x, c_y and c_s are constants. In order to apply the interpolation equation, the constants must be found. To do this, a special module is used by the GPU (the interp module; which is heavily pipelined to meet timing constraints), which uses the given three coordinates and their corresponding color or texture coordinate. Using these three equations and Cramer's rule to solve for c_x, c_y and c_s for each color (R, G and B) or the texture coordinate (u and v). Thus, there are a total of 5 of these interpolation modules near the GCMD to calculate the constants needed by the color and shader stages. These stages simply apply the above equation to determine the needed coordinate or color.

3.3.3 Testing

Verification of the GPU came in stages. The first stage was small unit tests of the line finding and triangle modules with a testbench that fed in coordinate data and printed to the terminal a grid of either hash marks or periods which clearly illustrated whether the proper shape was being drawn. Next came testing the whole GPU module, in pieces. Thus, a single command was tested and VCS was used to track down any problems by flowing through the data path and FSMs used by that instruction. Verification of the GPU consisted primarily of running short programs (of GPU commands) in simulation with a testbench that simulated VRAM. Periodically, the testbench would dump the contents of VRAM into a HEX file, on which a python script was run to convert the data into a PNG image. Viewing the image provided invaluable insight into what might be going wrong and illustrated the GPU could function in simulation. Once synthesized, SignalTap was used to debug any strange synthesis-related errors.

3.4 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.

We attempted to create the MDEC for this project, but due to time constraints and lack of documentation detailing some key hardware details of how commands are processes, the unit was abandoned. There is also reason to believe that it would not have fix on the board with all the other components as it relies on a good amount of multiplication.

3.5 Video

3.5.1 Adventures with 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 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.

Unfortunately, we were unable to get HDMI functioning during this project, but our description is provided here for posterity.

3.5.2 VRAM and Display

The video system in our implementation uses the on-board SRAM and VGA to display video to the screen. SRAM is used as VRAM for the GPU and a memory control module arbitrates access between it and the display module. When the display module needs to display new data, the memory controller disables the GPU's ability to perform memory accesses and dumps an entire row of the display area into a dual-ported row buffer. The display module reads out from this row buffer and send the colors to the VGA output via a DAC on the board. The display out module converts screen row coordinates to VRAM y coordinates and only requests a new row if the new y is different from that stored in the row buffer. The memory controller is responsible for converting the screen column coordinate (along with the y coordinate from the display module) into a VRAM address and loads up the row buffer completely. After this operation is complete, the memory controller gives back control of VRAM to the GPU, allowing it to perform reads and writes. Due to the timing of the VGA module, the display module runs on a 50 MHz clock while the memory controller runs at 33.3 MHz. We attempted to implement a more complex memory controller that would run at 100 MHz and would never block the GPU nor display module. However, timing issues plagued this implementation and it was abandoned in favor of the more synchronous design described above.

3.6 Memory

The PSX has several memory elements that can be accessed by the CPU, and by other components via DMA. The memory breakdown of the PSX is illustrated in the Memory Overview.

The primary memory chips are Main Memory, VRAM, and Sound Ram (SRAM). In addition there is also memory used by the CD-ROM controller as a buffer for data being read. The 32-bit addressing to Main Memory has three mirrors to the same physical memory 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 and CPU related caching. The PSX also

Table 3.2: Memory Overview

	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	0x00000000 to 0x001FFFFF	'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	0x80000000 to 0x801FFFFF	'Virtual memory' - mirrors KUSEG with caching enabled. KSEG2 addresses have a '100' prefix.
KSEG1	0xA0000000 to 0xA01FFFFF	'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 Control	1 + 32 k	Includes RAM, ROM, and buffer
Memory Cards	128 k	Extra memory slots that can be accessed through memory-mapped I/O.

contains several addresses of memory mapped I/O. Memory mapped I/O is used by the CPU to communicate with other pieces of hardware such as the GPU, SPU, controller peripherals, CD-ROM controller, etc. It also includes several registers that control internal functions such as timers and DMA. What would typically be the data cache in a standard MIPS is instead used as a Fast RAM, or scratchpad in the PSX. This region of memory can only be used for data and cannot be used to store code. The BIOS is stored in a separate 512KB ROM in the PSX and can be accessed using the last 512K addresses in each memory mirror. VRAM and SRAM are self-explanatory. They deal with their respective media outputs and processing. The GPU handles the framebuffer, texture pages and texture palettes in VRAM.

3.6.1 BIOS

The Altera board has a rather limited space in terms of memory elements. There is 2MB of SRAM, 432KB of blockram, and 128MB of SDRAM. Sound was entirely dropped from our project so we didnt have to worry about the 512KB of SRAM required by the PSX. For the BIOS we decided to use blockram because it has the option to be initialized to a prescribed value when synthesizing the design. The Quartus MegaFunction library provides a very nice interface to set up blockram of whatever size you want and prompts you to provide an initialization file. Using a BIOS dump we found online, we wrote a short Python script to produce a MIF file (used by Quartus). This was before we knew that the DE2-115 only has 432KB of blockram. This posed an interesting problem. Luckily, (using another python script) we found that a huge chunk of the BIOS we were working with was filled with all zeros; enough to bring the size down below 432KB. The Block RAM Arrangement shows how this shrinking was achieved.

Table 3.3: Block RAM Arrangement

BIOS	**Total Size**	512 KB
	Available Size	432 KB
The Breakdown		
	Number	**Address Range**
Blockram (256K)	-	0x00000 0x0FFFF
Blockram (16K)	0	0x10000 0x10FFF
	1	0x11000 - 0x11FFF
	2	0x12000 0x12FFF
[— **All Addresses in this range contain zeros** —]		
	3	0x19000 - 0x19FFF
	4	0x1A000 - 0x1AFFF
	5	0x1B000 - 0x1BFFF
	6	0x1C000 - 0x1CFFF
	7	0x1D000 - 0x1DFFF
	8	0x1E000 - 0x1EFFF
	9	0x1F000 - 0x1FFFF

3.6.2 Scratch Pad

The Scratch Pad was also located in blockram. Unlike the blockram used for the BIOS, the Scratch Pad was configured as RAM, not ROM. In the BIOS we only cared to read, so in the MegaFunction setup, we just created the necessary lines for reading and omitted the rest. For blockram configured as RAM, there are three additional inputs; data in, write enable, and a byte enable. The Scratch Pad only takes up 1KB, so it was able to fit in what was left of the blockram after stuffing the BIOS in.

3.6.3 Main Memory

Main Memory requires 2MB of physical address space. The only remaining body of memory that could fit that size was the SDRAM. Working with DRAM in general can be tricky. There is a tight protocol that has to be followed, and direct addressing is not an option. The chips are organized into banks, rows, and columns, and require precisely timed strobes of each of these signals to read and write from a desired address. Trying to generate the provided interface from the Quartus MegaFunction library was not leading to much success. There are a few other tools within quartus that can be used to generate SDRAM interfaces but many of those want you to also include a NIOS soft-processor in your project to manipulate the SDRAM controller. We were fortunate enough to get some help from the F12 Real Time Ray Tracer team and take their SDRAM module entirely and use it in our project without any problems. It should be noted that the qsys SDRAM module used in our project has been known to fail timing when the board is approaching the 80% capacity mark.

3.6.4 Hardware Registers

The PSX has approximately 200 hardware registers each with specific functions. A general description of each register can be found in the Appendix for Hardware Registers.

Looking at the table you can see that there are several broad categories of registers each dedicated to a specific function or a specific component such as the GPU or SPU. These registers provide a means for the CPU to communicate with the other components directly via memory reads and writes.

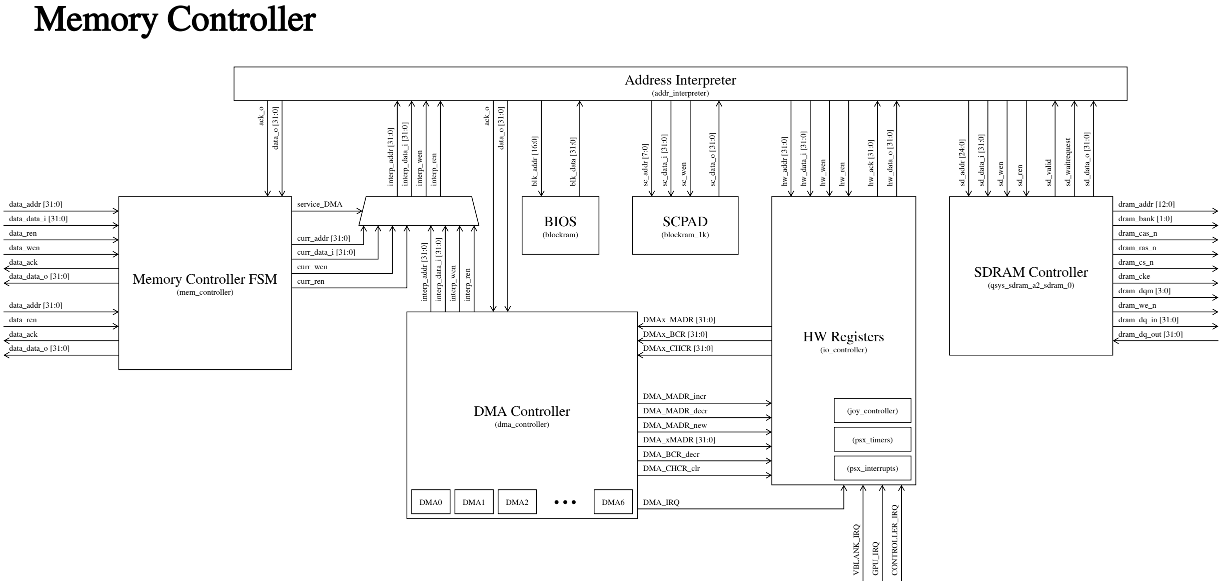
In our implementation of the PSX all hardware registers are handled through a single module (io_controller). This module with receive read and write requests from the CPU providing a fairly standard read-write-acknowledge interface to all the more complicated functions carried out by these registers. A read to a hardware register will latch the appropriate value and send it back to the CPU. When issuing a write, many of the hardware registers behave slightly differently. Some will trigger a reset of some counter. Others will only perform an AND operation with the existing value of that register. Some of the hardware registers, particularly those relating to the controller, are connected directly to external signals.

The addresses that behave as normal registers are handled at the top most level of the io_controller. These registers are declared and handle reads and writes directly through the FSM of the io_controller. For the more complicated registers, several submodules are implemented within the io_controller to facilitate custom actions. These specialized addresses include but are not limited to the GPU, SPU, MDEC, timers relating to vsyns and hsyncs, DMA control, and the controller interface.

3.6.5 Memory Controller & Address Interpreter

The memory controller FSM takes in memory requests from the CPU and is also aware of when DMA wants to do things. In the actual PSX, the CPU has two channels by which it can request a memory transfer. There is the instruction bus which is used to read the next instruction, and there is the data bus which is used to perform all other reads, writes, stack operations, etc. In the console it appears to be that Main Memory is properly dual-ported so these two channels can simultaneously read and write from memory (the instruction bus

Figure 3.5: Memory Controller



never does writes). However because of the limitations of the memory elements we used on the Altera board, there is no way to provide true dual-ported memory. Instead, our implementation did a dance between the three sources of memory requests; Instruction, Data, and DMA. The FSM interleaves memory accesses by ensuring that no single channel will be serviced twice before the other two are service if one of them are requesting a memory transfer. For example, if the Instruction line requests a read and both the Data line and DMA want to access memory, the next Instruction read will be stalled until both Data and DMA are serviced. The same rules apply for all three lines.

The heart of the memory controller is the address interpreter. From the memory controller FSM, the address interpreter takes in a 32-bit address and redirects the read/write to the appropriate physical memory. The first step of the process is to take the address and eliminate all of the mirroring bits. In our implementation we did not write the cache. This is mainly because we could not find any documentation whatsoever detailing the operation of the cache, and because caches are hard. So, with the mirroring bits dumped, the next step is to determine which physical memory the address is referring to. The mapping to physical memories is illustrated in the Memory Map table. The address interpreter FSM performs the read/write to the appropriate memory chip and then feeds back an acknowledge indicating the end of the transfer and in the case of a read, the data read. The CPU or DMA then takes the output of the address interpreter and the next memory request is serviced.

Testing the memory controller was done primarily using VCS and its GUI interface. In order to test interactions with physical memory chips we had to use memory models for simulation. Memory models were provided by Altera for SDRAM and blockram. In order to use the memory models for blockram, the necessary files have to be included into the project. They can be found in the simulation library in the Quartus installation on AFS. SDRAM had a similar simulation model provided by Altera however to use the memory model for SDRAM some tweaks had to be made to the HDL. We found that the memory model does not put data on the line the same way the actual chip does on the board. Once we were testing the BIOS on the actual FPGA hardware, the main tool for testing became SignalTap. At this point we were making changes to the code to match the behavior of the PSX and needed to see how the BIOS responded to our HDL.

3.7 DMA

To allow different components in the console to directly access Main Memory, the PSX uses DMA. There are seven DMA channels in the PSX listed in the DMA Mode table.

Each channel has three registers that control their function. The MADR (Base Address) register is used to write the start address of the DMA transfer. The BCR (Block Control) register is used to dictate the number of words that will be transferred and the size of the word. The CHCR (Channel Control) register is used to set up the direction of the transfer (to or from Main Memory), the step direction (forwards, or backwards), the transfer mode, and it is used to initiate a transfer by setting certain bits. Each of these DMA channels can operate in three modes.

Table 3.4: Memory Map

0x00000000-0x0000FFFF	Kernel (64K)
0x00010000 0x001FFFFF	User Memory (1.9 Meg)
0x1F000000-0x1F00FFFF	Parallel Port (64K)
0x1F800000-0x1F8003FF	Scratch Pad (1024 bytes)
0x1F801000-0x1F802FFF	Hardware Registers (8K)
0x80000000 0x801FFFFF	Kernel and User Memory Mirror (2 Meg) Cached
0xA0000000 0xA01FFFFF	Kernel and User Memory Mirror (2 Meg) Uncached
0xBFC00000-0xBFC7FFFF	BIOS (512K)

Table 3.5: DMA Mode

	Channel	Function	Supported Modes
0	MDECin	RAM to MDEC	1
1	MDECout	MDEC to RAM	1
2	GPU	lists and image data	1,2
3	CDROM	CD-ROM to RAM	0
4	SPU	???	???
5	PIO	???	???
6	OTC	Set up Ordering Table	0

3.7.1 Mode 0

Mode 0 is used strictly by channels 3 and 6 (CD-ROM and OTC). This mode behaves a little differently for the OTC and CD-ROM. With the OTC, what Mode 0 does is set up an Ordering Table. This is essentially an empty linked list of some prescribed size. At the start address, DMA will write the next address in memory. The next address will hold the address that follows it and so on. This write sequence is continued for the number of words declared in the BCR register. At the last address, the escape sequence 0x00FFFFFF is written to indicate the end of the linked list. For the CD-ROM, this mode is used to stream in CD-ROM data starting at some address set in the MADR register.

3.7.2 Mode 1

This mode is used to transfer blocks of data either to or from Main Memory. It is relatively straight forward. The block size and number of blocks is set in the BCR register. The system should not ever write a block size that is too big for the receiving components (i.e. the block size should never be greater than 0x10, 16-bits, for the GPU or SPU). The required number of blocks is transferred in 32-bit sections, and the transfer ends when the block count is decremented to zero.

3.7.3 Mode 2

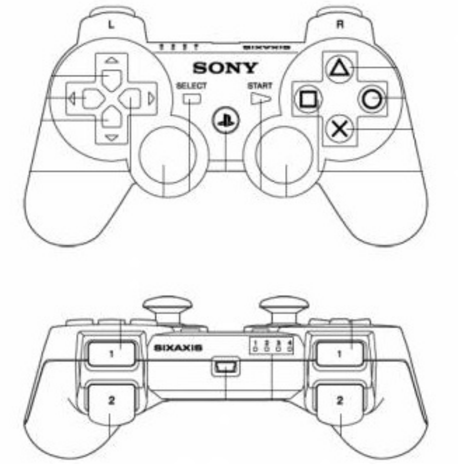
The GPU is the only component that uses Mode 2. This mode is used to traverse a linked list in memory and send instructions and data to the GPU. The start address will contain a value in the following format: 0xAABBBBBB, where 0xAA is the number of words stored in that entry, and 0xBBBBBB is the address of the next entry in the linked list. At each entry, DMA will send 0xAA words to the GPU and then move to the next entry. This process is repeated until 0xBBBBBB contains the escape sequence 0xFFFFFF. This is how DMA knows it has reached the end of the linked list and to stop the DMA transfer. DMA will only send the contents of each entry to the GPU. It uses the pointer information for itself to know where to look at next.

Should multiple DMA channels be active at the same time, the DMA Control Register (DPCR) is used to determine which channel has priority and that channel is serviced first to completion. Finally DMA interrupt requests are triggered upon completion of a transfer.

3.8 Controllers

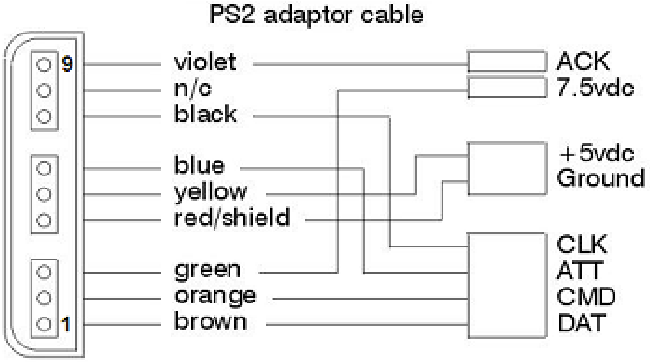
The standard digital controller of the PSX features 14 buttons, active-low, and uses 8-bit serial communication to talk to the game system. There are also other controller formats that include analog or digital joysticks (See Controller Diagram). These controller still talk to the system over the same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

Figure 3.6: Controller Diagram



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 Controller Pinout). 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 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 3.7: Controller Pinout



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.

The entire controller interface is handled through memory mapped I/O. The clock for the controllers is derived from a counter located in the JOY_STAT register. This timer decrements at 33MHz (the system clock speed) and upon reaching zero, it is reset to the reload value which is stored in JOY_BAUD. The counter elapses twice for each period of the controller clock; once for the clock set low and once for the clock set high. Under normal operation the reload value is set to 0x88 cycles, which results in a controller clock approximately equal to 250kHz. A transfer is initiated by the CPU when writing to the JOY_TX_DATA register. This register has the same address as where the RX_FIFO is stored (writes go to TX, and reads come from the FIFO). The controller ACK signal is connected directly to a bit of the JOY_STAT register and is typically held for about 100 system clock periods. It isn't entirely clear, but it seems to be that the ATT signal is also controlled by the CPU by a bit in JOY_CTRL. The CPU is entirely in control of interactions with the controller and is alerted of completed transactions via interrupts.

3.9 Game Data

The PSX uses CD-ROMS for games. The CD-ROM reader in the PSX is controlled by a small Motorola 8-bit CPU. This chip is connected to the main CPU through memory mapped I/O in a rather convoluted way described in the Appendix for CDROM Memory Map. Data from the disc is read out into a buffer and transferred to Main Memory by DMA. The Motorola chip is controlled by sending byte commands into a fifo. The CPU can know the status of the CD-ROM Controller by either reading the status register or receiving interrupts for each command sent. The CD-ROM controller in the PSX is also capable of playing audio CDs, but that feature was low in priority compared to playing games.

Having our system interface with physical game discs was thrown out as an option early on in the semester. Interfacing with an actual CD-ROM controller has several challenges and should be avoided is possible. There is a completed project called PSIO which is basically a piece of hardware that interprets 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. Data is then sent back to into the buffer from where DMA can ship the data to Main Memory. We tried to get in touch with the person behind this project, but we didnt have much luck. Instead we had to implement a similar interface ourselves. It should be noted that typical games for the PSX range from about 50MB to 700MB. This is the standard capacity of a CD-ROM. Some games go even further with multiple game discs resulting in a game worth over a gigabyte of data. The Virtex7 has 1GB of SDRAM, and would be able to fit an entire game locally, but no other board in the class has enough space to hold any reasonable PSX game. Also, should you want to have a game stored locally in SDRAM, the data still has to be ferried over to the board before a game can be played, and with such large ROMs, this doesnt seem like a good solution. Therefore the game ROMs should be stored in some form of flash memory that can be accessed by the FPGA. The first attempt was via SD Card.

SD Card IP Cores are available; there are a few on OpenCores. There is also an SD Card interface provided by Altera with Quartus. But, every single option available will require a soft-processor core to communicate with the card. This is understandable because a typical SD Card will be formatted in some file system such as NTFS, and having a soft-processor running embedded Linux would make it easy to read files from an SD Card. However should you want to read raw data from an SD Card, we could not find a single readily available native interface for SD Cards. Also, a quick search provided no easy way to load raw data onto an SD Card either, so this option was looking bad.

Round two involved a Raspberry Pi. Despite our efforts to avoid a soft-processor core and embedded programming, the only viable option to reading from flash memory involved C code and a separate processor. The Raspberry Pi was an attractive option because it has several GPIO pins that could be used to interface with the FPGA board. It has 17 GPIO pins. We used four pins for commands, eight pins for data, and one pin each for FPGA acknowledge and Raspberry Pi acknowledge. An additional pin was sued to provide a clock from the FPGA. The Raspberry Pi benchmarks claim to be able to read and write to GPIO using the native C interface at speeds up to 22 MHz, so we opted for 16.5MHz which is half of the system clock. We loaded the game ROM onto a flash drive and wrote a simple C program that would listen to the GPIO pins and serially receive the instructions that would normally be sent to the Motorola chip. Certain instructions relating to the CD-ROM motor could be ignored, but their response values still had to be faked, and the interrupts associated with those instructions still have to be fired to make the CPU think it is working with the real thing. The CD-ROM interface with the Raspberry Pi has yet to be tested thoroughly, but should serve as a good starting point should there be a lack of physical memory available on any given FPGA board.

4 Status

4.1 What is Done

In the end, we were unable to boot the BIOS. We were able to have a completely working GPU (stand alone), a working CPU/GTE and memory system, and the controller interface. Unfortunately, we ran out of time marching through the BIOS on the board with SignalTap. However, the CPU/GTE does execute commands fine, and is able to read from the BIOS and read, write and execute from main memory. The BIOS runs fine up to a point where it differs from the no$psx simulator and fails to draw the Sony logo and start screen. The GPU works well and is able to draw almost everything the original PSX GPU could. The controller interface is also solid and able to recognize all button presses correctly using the original digital controller.

4.2 What Needs to be Done

Quite a bit. As seen above, a variety of components we did not even get a chance to work on. The core system (CPU/GTE, GPU, Memory, Controller) is largely complete (though not bug-free). However, the CDROM and MDEC are far from complete and the sound system was not even attempted. Unfortunately, time and HDMI makes fools of us all.

If you are continuing from where we left off, synthesize the project, upload to the board, open up no$psx, and start back tracing. We are in the middle of the BIOS and entering an (invalid?) loop right before the white screen comes on in the BIOS animation. There are still some synthesis quarks in the GPU to deal with and potentially more features to add to the CPU before a full successful boot can happen. Simulation will take unbelievably long, and it is worth the 20+ minutes of simulation to use a logic analyzer.

As stated under the SPIM section of tools, Pong is a good way to test CPU, memory, and GPU integration. Under team_psx/psx/cpu/mips_w_tb/dat_files, you will find a version of extremely buggy pong that was hacked in under an hour in an attempt to demo the system. Aside from Pong being broken, the GPU also is reading 0s even though the right commands are being fed. But the file is small enough that it will work in simulation.

5 Words of Wisdom

5.1 Don't Take on Two Major Challenges at Once

At the beginning of the semester we had the impression that HDMI would be a quick and easy project to write an interface for. Little did we know that HDMI constitutes an entire semester-long project on its own. We spent over a month fighting against the Virtex-7 board when we could have been making progress on the PSX. This also applies to most features of evaluation boards because everyone loves soft-core processors. Beware of soft-cores; they won't help you in this course.

5.2 Follow a Nice Style Guide

The key to synthesis is good style. It also makes integration a lot less painful. As Mike will tell you many times over, the 18-240 FSM style (current state assignments register in always_ff, output logic and next state in always_comb) will always work and keep you out of trouble. Watch out for latches inferred.

5.3 Interfacing with the Outside is Not Reliable

When you are on a time crunch, relying on outsiders (*cough* AnalogDevices *cough*) to respond to your forum post is probably not the way to get things done.

5.4 Selections from our "Stupid Things" Document

- Make sure project paths are not too long when using debug cores in Vivado. It will error at opt_design otherwise.
- Local copies of Vivado run great, but make sure to "sudo" in order to use board facilities. This goes for any local install of any tools on a Linux lab machine.

- If you are using Quartus, don't create bidirectional logic anywhere but the top module. Don't propagate inouts through modules, keep them only at the top.

- When using always_ff with asynchronous reset, Quartus will attempt to infer latches on you unless the whole signal is assigned in more than one place.

  For example, this will infer latches:

  ```
  always_ff @(posedge clk, negedge rst)
      if (rst) cop <= 2'b0;
      else if (something) cop[0] <= nxt_cop[0];
      else cop[1] <= nxt_cop[1];
  ```

  This will not:

  ```
  always_ff @(posedge clk, negedge rst)
      if (rst) cop <= 2'b0;
      else cop <= nxt_cop;
  ```

- Synthesis will do dumb things and sometimes writing it a different way will make it better. The document in the repository has a list of random fixes, such as using always_ff instead of always and moving a nested "else" "if" to just "else if" and using always_comb instead of assign, that make synthesis work like simulation, but for no clear reason.

6 For Future Iterations

Use our code. Everything you need to synthesize is under team_psx/psx/system. Open up the system_top project with Quartus and you're good to go. To speed up synthesis make sure to turn on "Smart Compilation", utilize all cores, and basically just turn on/off anything that will shave synthesis time. The repository is a bit messy, but the directory with the synthesizable project contains the most up to date files. Outside in the other subdirectories are individual components and (potentially) facilities to test it. Even if you just use it as a guide for writing your FPGA PlayStation, it's a lot better than nothing (which, by the way, unlike the GameBoy or Sega Genesis, there really is nothing).

Get all the documentation you can. Our repository contains a lot of what you will need. Be sure to get the Psy-Q documents (from psxdev.net) and look over them, they have pieces of information that is quite good and occasionally lacking from the Nocash and Everything documents (see Acknowledgements). Also, we made a few useful tables and such in the Appendices.

For the adventurous individual(s) who want to take a stab at the Virtex-7 read the following carefully. If you want to use HDMI, there are a few steps of configuration that need to be done before the ADV7511 chip can be used. Configuration of the ADV7511 is handled over the I2C bus on the VC707. The I2C master is the FPGA chip. Between the FPGA and the ADV7511, there is a bus switch (PCA9458) that has to be configured first before the I2C bus will be able to talk to the HDMI chip. If you do not configure this switch first, none of your I2C commands will be heard. Once you are able to talk to the ADV7511 chip, you need to configure the chip to output HDMI. Guides for configuration can be found on AnalogDevices's website, but some information is misleading. First, note that almost all help will suggest that you use a soft-processor core loaded with Linux and C code to communicate with the ADV7511. Also, there is one register in particular on the ADV7511 that is responsible for switching the chip between DVI and HDMI modes. We were never able to set the chip in HDMI mode. Reading back from this register would always indicate that the chip was in DVI mode, so it may not even be possible to output sound from the

VC707. Note that even with the configuration mostly working, we were not able to output to the display via HDMI. We are not sure why. We believe it was because we didn't have the timing right, but there might have been another reason. A second word of caution is that the Virtex-7 only has 8 readily available GPIO pins located in its XADC port. If you are planning on using the VC707 for a project that will require several GPIO pins, please reconsider or do some research on the expansion boards for the Virtex-7.

7 Personal Statements

Demo Day, December 6, 2013, 8 AM

7.1 Mike Rosen

Even though we were not able to complete the full working system, the project itself was a rewarding and fun experience. Flowing through the process of conception, design, implementation and integration is very educational and certainly gave me a better understanding of how to work in a team building a system you individually would have no hope of completing (even 3 of us were not able to do it). Helping my teammates debug modules I only tangentially understood was an interesting experience, and getting help from them was very useful even though they did not fully understand what I was working on (the PlayStation is large enough that this was a common occurrence).

In our HDMI attempt, I was responsible for the audio portion. Implementing the SPDIF protocol was not too difficult. However, due to the I2C complexity of the VC707 board and AnalogDevices not giving up the timing information we needed, we had to abandon HDMI and the VC707 all together. Unfortunately, this toolset switch was the biggest factor towards our inability to finish the project. While the Virtex-7 is a powerful chip, the VC707 is a terrible board of this class. The lack of GPIO and reasonable audio-video capabilities makes the board horrible for the typical project in this course. If we were ambitious enough to complete our HDMI interface, the board might become usable.

In terms of the PSX system, I was responsible for the graphics system, including the GPU and MDEC. Unfortunately, due to time constraints and lack of documentation on the hardware, the MDEC was abandoned. The GPU was plenty of work for the semester and became the only thing we could demo due to integration problems. Building the PSX GPU was a lot of fun. Due to the lack of documentation, I pretty much just had an “ISA” and a few vague pieces of information on some of the internals. Beyond that, the design of the original was completely a mystery, which allowed me a lot of design freedom. Starting off, I had no idea how to do anything in terms of the graphics functionality. The shading, rasterization and texture mapping used by the PSX were things I had no previous experience with. Learning how to do these things by talking to friends, other professors and consulting the internet was great practical experience. Much time was spent pondering how exactly to implement all the interesting graphics functions the GPU could do. And the results were worth it. When my simulation produced the same color gradient triangle as the OpenGL teams, I was very happy. Soon, I was drawing things that looked like they came out of the original PSX.

However, synthesis proved to be a huge problem of its own. Switching to the Altera board seemed a good choice at first. The SystemVerilog support and our past experience with it made it a better choice than the relatively-unknown Virtex-5. Unfortunately, while the size of the chip is roughly the same in terms of number of logic elements, the constraints of the board became a major roadblock. To implement VRAM, I needed to use the SRAM on the board (which at first seemed perfect). I spent a lot of time (2-3 weeks) determining cool ways to get over the limitation that the SRAM chip was single ported and I needed to access VRAM for the GPU to draw and the VGA to display the image. However, in the end, my virtual dual-porting attempted did not work so I had to change my design to block the GPU when VGA needed to send pixels to the screen. Following that was the issue of board resources. My first synthesis that actually completed used about 160% of the LEs on the board. So I had to completely eliminate the parallel pipeline to a width of 1. Another big problem was timing. The GPU did some rather complex operations and timing of critical paths become a limiting factor in my design. TimingAnalyzer told me that my synthesized GPU had a maximum frequency of 4.9 MHz rather than the need 33.3 MHz. Excessive pipelining fixed this issue, but a more powerful board would not require so much effort to optimize.

The last week, we spent living in the lab. I spent 31.5 hours in Hamerschlag without even leaving the building once. The sun became a foreign concept. On the Friday after the demo (once it was all done), I took a 3 hour nap, and soon after went to sleep for 14 hours straight, and snoozed for another 3 hours. Just be aware, this is what it takes to do the more complicated projects, especially those never before attempted (not even by anyone on the Internet...). Listen to Paul (F12; Team Real Time Ray Tracer) and I, **a big project in this class will be your life this semester**. And even though we did not get the BIOS working like we wanted (we were soooo close), it was a very worthwhile experience (not the lack of sleep; the project!). Now that we've opened the door on the PSX, another group SHOULD be able to complete it (if they are willing and able; but good luck!). Any group that wants to go on to the N64 (or another more modern system), I recommend to Professor Nace or any future professor to NOT LET THEM TRY (unless the toolchain changes to let more complex systems be implementable).

What I would recommend to future groups is the following:

- Tackle only ONE major challenge: For us, HDMI and the PSX were essentially 2 projects in 1 semester and the former ate up too much time that we did not get to finish the latter.

- Start early: We did and we still did not finish...

- Be careful of complexity: The PSX is certainly a possible project for this course. You have to be willing to do what we did and then some; and not work on any other major challenge (HDMI). It is a bunch of simple machines after all. But there are a bunch more of them than most other projects done in this course and they interact in more complicated ways.

- Documentation must be thoroughly scraped: While working on this project, we would sometimes find out something we needed to know weeks ago in some obscure documents. Knowing everything about your system in the design phase (before code) is best (though not always possible).

- Pick something fun: If you are not having fun, change projects or drop the course.

7.2 Anita Zhang

My main contributions to the PlayStation project were CPU modification and integration, the GTE (coprocessor 2), system integration, and testing the complete system. I looked over all my old status reports to try and figure out why the whole project seemed so back-loaded and I noticed that it was not until about week seven that we started moving away from HDMI. In regards to HDMI, which took up a major part of the first semester, I worked on the video interface, as well as making the software interface "work".

With regards to time spent, the first half of the semester when we were still working on HDMI or starting separate components, I recall being in lab on Tuesdays from 6PM to midnight, and Sundays from 3PM to midnight. This was both HDMI anything and CPU/GTE research for me. When we started working asynchronously (about week seven) I recall working mostly on Sundays from 6PM to midnight on GTE research, implementation, and testing. During midterms not much PlayStation work was done at all. About week ten I started going to lab more to integrate the CPU and do system testing (it helped with productivity too). A couple of weeks after, during system integration, lab time spiked upwards, and from Thanksgiving to Demo Day I was in lab debugging from 12 to 20 hours a day.

Silly thing about debugging, the day before the demo we were debugging the BIOS and hit another point where the CPU was not writing the right value into EPC (exception program counter), compared to the no$psx debugger. At the time we were all so sleep deprived and did not realize that EPC was just a pointer to where it was supposed to return in the instruction stream. We spent 2 hours trying to figure out how it got that value written in because we usually back trace if we are stuck. The silly thing was an hour before that I told Arnob EPC gets written with an address during exceptions! That week was the most hellish I have ever experienced. I am pretty sure no one in our group slept more than 3 hours a day.

I had a couple of job interviews during the project and I talked to one of the interviewers about Team PSX. The interviewer asked me if I regret taking on the PlayStation in favor of Defender (our “Plan B”), a definitely finish-able project in one semester. I thought about it and told him no. This is might be the biggest project I have worked on (close competitors include the 15-410 kernel and 18-447 MIPS) and I am too grateful for working with such passionate team members. I admit I was doubtful during our first weeks, but by the time we had to make our “Defender or PlayStation” decision I was not willing to leave PlayStation just because there was doubt of finishing. And (I may be stealing this from Mike because he said it first) the lack of detailed documentation in some of the components really opened the way for individual design. It is nice to have exact details and pre-existing components, but implementing your own design is the funnest way to go. And there is no point to a project course if it is not fun! And challenging! For that reason I enjoy pooling a lot of time into projects. I regret not being to work on more parts of the system since our roles were very specific. And I am still really disappointed at not being able to load the BIOS and see the startup animation. That is probably the one thing I will never let go (no matter how much it is about the journey)!

The combination of this course and 18-341 has made me realize Verilog and its tools are so unbelievably unpredictable sometimes that I cannot do this for the rest of my life. Take that as you will.

In terms of class structure, LCD lab was useful for getting us started with the tools, but the LCD itself was never used after that. Some teams never make it to sound, and some do not even have sound, so I am in favor of wiping sound lab. 10/10 would keep ChipScope lab. I wish design review did not take up so much time, but I enjoyed the free nature of the course.

Past projects reflect this, but future students should not attempt to use the HDMI on the Virtex-7 until someone has a working hardware interface. AnalogDevices should get punched for that. Also, taking on more than one heavy load in this course (HDMI and

PlayStation, in our case) should be interrupted and stopped by week two. But hopefully we have left enough information for iteration two of the PlayStation!

7.3 Arnob Mallick

I would like to start off by stating that 18-545 Advanced Digital Design has the potential to be one of the most rewarding undergraduate classes for anyone who should choose to take it. Even though we didnt complete our project, and barely had anything prepared to demo on the last day of class, the entire experience as a whole was invaluable. Our team decided to try to recreate the original PlayStation. Yes, it was a pretty crazy, but when the idea was put on the table you could immediately tell that all three of us were mentally salivating on the inside at the thought of such a cool project (well at least I was). I was in charge of a lot of almost everything that wasnt CPU or GPU. My first task was to get HDMI configured and working I believe our report documents our struggles with HDMI fairly well. It was an incredibly frustrating experience, but I suppose it could have been worse had I saved HDMI for later in the semester. For a console gaming system, or anything remotely game related that will require GPIO pins and video output directly from the FPGA board, the Virtex7 is NOT the right choice. I found myself in a hopeless cycle going back and forth with representatives from AnalogDevices to try and figure out what was going on with their abominable chip. I spent over a month doing this. During this time I threw together a controller interface that links all button presses on the controller with LEDs on the board.

Once we finally switched boards I started working on components of the actual PSX. From this point forward I worked on the memory controller for the CPU and DMA. This involved interfacing with a few different physical memories on the Altera board and creating all of the special registers used for memory mapped I/O. By the time I was completely detached from HDMI and settled using the new board I was able to gradually churn out a full memory interface. My main challenge for this course from start to finish was figuring out how to deal with peripherals. Doing things entirely within the FPGA is nice and predictable in most situations. Once you try to interface with SDRAM, or anything outside of the FPGA everything is a mystery. Good documentation is often hard to find and although there are often implementations online that are tried and tested, they will still require some degree of reverse engineering to read their (bad) documentation and configure everything to work with your board. Once I sorted out all my issues with external chips and such, I cant say I had any more major hitches. I implemented the memory interface incrementally adding support for more and more memory related interactions with the CPU. We found several bugs in the process, but since everything was isolated within the FPGA, I was able to trace those issues with ease and fix what I had to.

It is hard to say how much time was spent on this project in total. Almost always at least one member of three was in the 18545 lab. What I can say however is that if a project is on the same level as the PlayStation, the time spent working on 18545 related things should overshadow all other courses by some margin. Straight through Thanksgiving to the last week of the semester I spent between 12 to 20 hours a day working on the PSX. And several of those days were spent entirely in the 18545 lab locked up in Hammerschlag, especially the last few days.

As I mentioned earlier the class as a whole is great. I really enjoyed the self-paced nature of the course. This allowed me to allocate time and work around my other classes as necessary. However, this freedom should be taken with caution. Being able to move around time for 18545, doesnt mean you can push the entire project to the end of the semester. My team took a day to outline general deadlines which was certainly helpful to prevent nasty cases of procrastination. My impression of the course was also greatly affected by my teammates. Anita and Mike were awesome people to work with and I cant say we faced any real group issues. We all held each other accountable for their parts of the project and we all respected each other enough to try not to fall behind schedule too badly (We did eventually fall behind schedule really badly, but we were all on the same page at that point). I could have come out of this class with a mangled friendship and two new mortal enemies, but Im glad I was able to make a new friend and keep the one I already had. I think I learned a lot about working on a large project with multiple people. I learned about time management, and how to scale that up to larger groups. I also learned a lot from the other groups around us. I saw examples of personalities that were destructive, and situations that were not handled very well. I also saw people working very well together. Having everyone together in one lab is very useful for that reason. It gave me exposure to four other micro-communities that each had their own successes and failures.

To improve the class I would suggest adding to the disclaimer list of the Virtex7. When we started the semester, we knew that the board was new and that there wasnt much to work with yet. Now that a few teams have fought that battle against Xilinx, we can better prepare future teams for the challenges they will face. I think some of the labs at the beginning of the semester should be restructured. ChipScope (Xilinx) and SignalTapII (Altera) need to get more attention. There were people in the class, my teammates included, that still didnt fully understand how to take advantage of the probing tools. We would not have been able to get the BIOS working at all if we didnt know how to use SignalTap. Also regarding the lab machines. Those things are some of the most frustrating boxes ever. I had the impression that they were our machines to use as we wished, but this really isnt true. If we want to have all the tools on AFS, we have to be on the school network, and to be on the school network means dealing with all the permissions nonsense. I couldnt transfer directories of any appreciable size because for some reason it would route through my AFS space first which I havent bothered to increase in size yet. Which results in a machine with 500GB of disk space that can really only use a tiny amount at any one time. We need the installation discs and full access to these machines away from AFS. That way we wont be losing drivers every other month and all issues we face would be entirely our faults.

We got so close to drawing the PlayStation boot screen. We were literally only a few hundred, maybe thousand, instructions away from seeing the screen light up. I am looking forward to another team picking up from where we left off and finishing the PSX. Even with all the work that has already been done there is still a lot we couldnt finish and it would certainly put the biggest smile on my face to see our project taken to completion.

8 Acknowledgements

Thanks Bill Nace for letting us run with our crazy ideas. Feel free to let anyone use our code and to post our report for future generations.

Thanks to Astav Sacheti and Apple for sponsoring the class and understanding the rigor of our project. It really helped lift us from our defeated states and put a smile on our faces after a long struggle. And thanks for the iPods!

Special thanks to Martin Korth (http://nocash.emubase.de), the sole, broke/poor, German creator of no$psx and various other console emulators/debuggers under the Nocash name. May he get many donations so he can continue to better his debugger and not have a "real" job.

Thanks to Ryan MacDonald for providing us with a working SDRAM interface for the Altera DE2-115 board.

Thanks to Professor Kayvon Fatahalian for explaining interpolation to us. Without him, the GPU would not have been possible. Take his classes; he's awesome.

Thanks to Joshua Walker for compiling some very good information in his "Everything You Have Always Wanted to Know about the Playstation But Were Afraid to Ask" document. It really kickstarted this project and made us think we had a chance (which we did in the end).

And thanks to the PSXDEV community for answering some critical questions, as well as providing us with and pointing us to some of the best documentation on the internet! (http://www.psxdev.net/) Finally, thanks to Sony for making this console; its a really cool piece of hardware. We hope that you will release all the secrets; we are dying to know.

A GPU Tables

The following contains a full list of GP0 and GP1 commands for the GPU.

A.1 GP0 Command List

Table A.1: GP0 Commands

Opcode	Command Name	Description
0x00	NOP	Does nothing (No space in FIFO in PSX, does in ours)
0x01	CLR_CACHE	Clears Texture Cache
0x02	FILL_VRAM	Fills a rectangle in VRAM with given color
0x03	NOP (FIFO)	Does nothing (but takes up FIFO space)
0x04	NOP	Mirror 0x00
0x05	NOP	Mirror 0x00
0x06	NOP	Mirror 0x00
0x07	NOP	Mirror 0x00
0x08	NOP	Mirror 0x00
0x09	NOP	Mirror 0x00
0x0A	NOP	Mirror 0x00
0x0B	NOP	Mirror 0x00
0x0C	NOP	Mirror 0x00
0x0D	NOP	Mirror 0x00
0x0E	NOP	Mirror 0x00

Continued on next page

Table A.1 – continued from previous page

Opcode	Command Name	Description
0x0F	NOP	Mirror 0x00
0x10	NOP	Mirror 0x00
0x11	NOP	Mirror 0x00
0x12	NOP	Mirror 0x00
0x13	NOP	Mirror 0x00
0x14	NOP	Mirror 0x00
0x15	NOP	Mirror 0x00
0x16	NOP	Mirror 0x00
0x17	NOP	Mirror 0x00
0x18	NOP	Mirror 0x00
0x19	NOP	Mirror 0x00
0x1A	NOP	Mirror 0x00
0x1B	NOP	Mirror 0x00
0x1C	NOP	Mirror 0x00
0x1D	NOP	Mirror 0x00
0x1E	NOP	Mirror 0x00
0x1F	IRQ	Sets interrupt request bit
0x20	POLY_F3	Monochrome, Opaque 3-sided Polygon
0x21	POLY_F3	Mirror 0x20
0x22	POLY_F3S	Monochrome, Semi-transparent 3-sided Polygon
0x23	POLY_F3S	Mirror 0x22
0x24	POLY_FT3	Textured, Blended, Opaque 3-sided Polygon
0x25	POLY_FT3R	Textured, Raw, Opaque 3-sided Polygon

Continued on next page

Table A.1 – continued from previous page

Opcode	Command Name	Description
0x26	POLY_FT3S	Textured, Blended, Semi-transparent 3-sided Polygon
0x27	POLY_FT3RS	Textured, Raw, Semi-transparent 3-sided Polygon
0x28	POLY_F4	Monochrome, Opaque 4-sided Polygon
0x29	POLY_F4	Mirror 0x28
0x2A	POLY_F4S	Monochrome, Semi-transparent 4-sided Polygon
0x2B	POLY_F4S	Mirror 0x2A
0x2C	POLY_FT4	Textured, Blended, Opaque 4-sided Polygon
0x2D	POLY_FT4R	Textured, Raw, Opaque 4-sided Polygon
0x2E	POLY_FT4S	Textured, Blended, Semi-transparent 4-sided Polygon
0x2F	POLY_FT4RS	Textured, Raw, Semi-transparent 4-sided Polygon
0x30	POLY_G3	Gouraud Shaded, Opaque 3-sided Polygon
0x31	POLY_G3	Mirror 0x30
0x32	POLY_G3S	Gouraud Shaded, Semi-transparent 3-sided Polygon
0x33	POLY_G3S	Mirror 0x32
0x34	POLY_GT3	Gouraud Shaded, Textured, Blended, Opaque 3-sided Polygon
0x35	POLY_FT3R	Strange Mirror 0x25
0x36	POLY_GT3S	Gouraud Shaded, Textured, Blended, Semi-transparent 3-sided Polygon
0x37	POLY_FT3RS	Strange Mirror 0x27
0x38	POLY_G4	Gouraud Shaded, Opaque 4-sided Polygon
0x39	POLY_G4	Mirror 0x30
0x3A	POLY_G4S	Gouraud Shaded, Semi-transparent 4-sided Polygon

Continued on next page

Table A.1 – continued from previous page

Opcode	Command Name	Description
0x3B	POLY_G4S	Mirror 0x32
0x3C	POLY_GT4	Gouraud Shaded, Textured, Blended, Opaque 4-sided Polygon
0x3D	POLY_FT4R	Strange Mirror 0x2D
0x3E	POLY_GT4S	Gouraud Shaded, Textured, Blended, Semi-transparent 4-sided Polygon
0x3F	POLY_FT4RS	Strange Mirror 0x2F
0x40	LINE_F2	Monochrome, Opaque Line
0x41		
0x42	LINE_F2S	Monochrome, Semi-transparent Line
0x43		
0x44		
0x45		
0x46		
0x47		
0x48	LINE_FP	Monochrome, Opaque Poly-line
0x49		
0x4A	LINE_FPS	Monochrome, Semi-transparent Poly-line
0x4B		
0x4C		
0x4D		
0x4E		
0x4F		
0x50	LINE_G2	Gouraud Shaded, Opaque Line
0x51		

Continued on next page

Table A.1 – continued from previous page

Opcode	Command Name	Description
0x52	LINE_G2S	Gouraud Shaded, Semi-transparent Line
0x53		
0x54		
0x55		
0x56		
0x57		
0x58	LINE_GP	Gouraud Shaded, Opaque Poly-line
0x59		
0x5A	LINE_GPS	Gouraud Shaded, Semi-transparent Poly-line
0x5B		
0x5C		
0x5D		
0x5E		
0x5F		
0x60	TILE	Monochrome, Opaque, Variable-size Rectangle
0x61		
0x62	TILE_S	Monochrome, Semi-transparent, Variable-size Rectangle
0x63		
0x64	SPRT	Textured, Blended, Opaque, Variable-size Rectangle
0x65	SPRT_R	Textured, Raw, Opaque, Variable-size Rectangle
0x66	SPRT_S	Textured, Blended, Semi-transparent, Variable-size Rectangle
0x67	SPRT_RS	Textured, Raw, Semi-transparent, Variable-size Rectangle
0x68	TILE_1	Monochrome, Opaque, 1x1 Rectangle
0x69		

Continued on next page

Table A.1 – continued from previous page

Opcode	Command Name	Description
0x6A	TILE_1S	Monochrome, Semi-transparent, 1x1 Rectangle
0x6B		
0x6C	SPRT_1	Textured, Blended, Opaque, 1x1 Rectangle
0x6D	SPRT_1R	Textured, Raw, Opaque, 1x1 Rectangle
0x6E	SPRT_1S	Textured, Blended, Semi-transparent, 1x1 Rectangle
0x6F	SPRT_1RS	Textured, Raw, Semi-transparent, 1x1 Rectangle
0x70	TILE_8	Monochrome, Opaque, 8x8 Rectangle
0x71		
0x72	TILE_8S	Monochrome, Semi-transparent, 8x8 Rectangle
0x73		
0x74	SPRT_8	Textured, Blended, Opaque, 8x8 Rectangle
0x75	SPRT_8R	Textured, Raw, Opaque, 8x8 Rectangle
0x76	SPRT_8S	Textured, Blended, Semi-transparent, 8x8 Rectangle
0x77	SPRT_8RS	Textured, Raw, Semi-transparent, 8x8 Rectangle
0x78	TILE_16	Monochrome, Opaque, 16x16 Rectangle
0x79		
0x7A	TILE_16S	Monochrome, Semi-transparent, 16x16 Rectangle
0x7B		
0x7C	SPRT_16	Textured, Blended, Opaque, 16x16 Rectangle
0x7D	SPRT_16R	Textured, Raw, Opaque, 16x16 Rectangle
0x7E	SPRT_16S	Textured, Blended, Semi-transparent, 16x16 Rectangle
0x7F	SPRT_16RS	Textured, Raw, Semi-transparent, 16x16 Rectangle
0x80	CPYRECT_V2V	Copy pixels from a rectangle in VRAM to another part of VRAM

Continued on next page

Table A.1 – continued from previous page

Opcode	Command Name	Description
0x81	CPYRECT_V2V	Mirror 0x80
0x82	CPYRECT_V2V	Mirror 0x80
0x83	CPYRECT_V2V	Mirror 0x80
0x84	CPYRECT_V2V	Mirror 0x80
0x85	CPYRECT_V2V	Mirror 0x80
0x86	CPYRECT_V2V	Mirror 0x80
0x87	CPYRECT_V2V	Mirror 0x80
0x88	CPYRECT_V2V	Mirror 0x80
0x89	CPYRECT_V2V	Mirror 0x80
0x8A	CPYRECT_V2V	Mirror 0x80
0x8B	CPYRECT_V2V	Mirror 0x80
0x8C	CPYRECT_V2V	Mirror 0x80
0x8D	CPYRECT_V2V	Mirror 0x80
0x8E	CPYRECT_V2V	Mirror 0x80
0x8F	CPYRECT_V2V	Mirror 0x80
0x90	CPYRECT_V2V	Mirror 0x80
0x91	CPYRECT_V2V	Mirror 0x80
0x92	CPYRECT_V2V	Mirror 0x80
0x93	CPYRECT_V2V	Mirror 0x80
0x94	CPYRECT_V2V	Mirror 0x80
0x95	CPYRECT_V2V	Mirror 0x80
0x96	CPYRECT_V2V	Mirror 0x80
0x97	CPYRECT_V2V	Mirror 0x80
0x98	CPYRECT_V2V	Mirror 0x80

Continued on next page

Table A.1 – continued from previous page

Opcode	Command Name	Description
0x99	CPYRECT_V2V	Mirror 0x80
0x9A	CPYRECT_V2V	Mirror 0x80
0x9B	CPYRECT_V2V	Mirror 0x80
0x9C	CPYRECT_V2V	Mirror 0x80
0x9D	CPYRECT_V2V	Mirror 0x80
0x9E	CPYRECT_V2V	Mirror 0x80
0x9F	CPYRECT_V2V	Mirror 0x80
0xA0	CPYRECT_C2V	Copy data from a rectangle from Main Memory to VRAM
0xA1	CPYRECT_C2V	Mirror 0xA0
0xA2	CPYRECT_C2V	Mirror 0xA0
0xA3	CPYRECT_C2V	Mirror 0xA0
0xA4	CPYRECT_C2V	Mirror 0xA0
0xA5	CPYRECT_C2V	Mirror 0xA0
0xA6	CPYRECT_C2V	Mirror 0xA0
0xA7	CPYRECT_C2V	Mirror 0xA0
0xA8	CPYRECT_C2V	Mirror 0xA0
0xA9	CPYRECT_C2V	Mirror 0xA0
0xAA	CPYRECT_C2V	Mirror 0xA0
0xAB	CPYRECT_C2V	Mirror 0xA0
0xAC	CPYRECT_C2V	Mirror 0xA0
0xAD	CPYRECT_C2V	Mirror 0xA0
0xAE	CPYRECT_C2V	Mirror 0xA0
0xAF	CPYRECT_C2V	Mirror 0xA0

Continued on next page

Table A.1 – continued from previous page

Opcode	Command Name	Description
0xB0	CPYRECT_C2V	Mirror 0xA0
0xB1	CPYRECT_C2V	Mirror 0xA0
0xB2	CPYRECT_C2V	Mirror 0xA0
0xB3	CPYRECT_C2V	Mirror 0xA0
0xB4	CPYRECT_C2V	Mirror 0xA0
0xB5	CPYRECT_C2V	Mirror 0xA0
0xB6	CPYRECT_C2V	Mirror 0xA0
0xB7	CPYRECT_C2V	Mirror 0xA0
0xB8	CPYRECT_C2V	Mirror 0xA0
0xB9	CPYRECT_C2V	Mirror 0xA0
0xBA	CPYRECT_C2V	Mirror 0xA0
0xBB	CPYRECT_C2V	Mirror 0xA0
0xBC	CPYRECT_C2V	Mirror 0xA0
0xBD	CPYRECT_C2V	Mirror 0xA0
0xBE	CPYRECT_C2V	Mirror 0xA0
0xBF	CPYRECT_C2V	Mirror 0xA0
0xC0	CPYRECT_V2C	Copy pixels from a rectangle of VRAM to Main Memory
0xC1	CPYRECT_V2C	Mirror 0xC0
0xC2	CPYRECT_V2C	Mirror 0xC0
0xC3	CPYRECT_V2C	Mirror 0xC0
0xC4	CPYRECT_V2C	Mirror 0xC0
0xC5	CPYRECT_V2C	Mirror 0xC0
0xC6	CPYRECT_V2C	Mirror 0xC0

Continued on next page

Table A.1 – continued from previous page

Opcode	Command Name	Description
0xC7	CPYRECT_V2C	Mirror 0xC0
0xC8	CPYRECT_V2C	Mirror 0xC0
0xC9	CPYRECT_V2C	Mirror 0xC0
0xCA	CPYRECT_V2C	Mirror 0xC0
0xCB	CPYRECT_V2C	Mirror 0xC0
0xCC	CPYRECT_V2C	Mirror 0xC0
0xCD	CPYRECT_V2C	Mirror 0xC0
0xCE	CPYRECT_V2C	Mirror 0xC0
0xCF	CPYRECT_V2C	Mirror 0xC0
0xD0	CPYRECT_V2C	Mirror 0xC0
0xD1	CPYRECT_V2C	Mirror 0xC0
0xD2	CPYRECT_V2C	Mirror 0xC0
0xD3	CPYRECT_V2C	Mirror 0xC0
0xD4	CPYRECT_V2C	Mirror 0xC0
0xD5	CPYRECT_V2C	Mirror 0xC0
0xD6	CPYRECT_V2C	Mirror 0xC0
0xD7	CPYRECT_V2C	Mirror 0xC0
0xD8	CPYRECT_V2C	Mirror 0xC0
0xD9	CPYRECT_V2C	Mirror 0xC0
0xDA	CPYRECT_V2C	Mirror 0xC0
0xDB	CPYRECT_V2C	Mirror 0xC0
0xDC	CPYRECT_V2C	Mirror 0xC0
0xDD	CPYRECT_V2C	Mirror 0xC0
0xDE	CPYRECT_V2C	Mirror 0xC0

Continued on next page

Table A.1 – continued from previous page

Opcode	Command Name	Description
0xDF	CPYRECT_V2C	Mirror 0xC0
0xE0	NOP	Mirror 0x00
0xE1	DR_MODE	Sets various drawing settings
0xE2	DR_TWIN	Sets the texture window
0xE3	DR_AREA_TL	Sets the drawing area's top-left corner
0xE4	DR_AREA_BR	Sets the drawing area's bottom-right corner
0xE5	DR_OFFSET	Sets the drawing offset
0xE6	DR_MASK_SET	Sets the drawing mask settings
0xE7	NOP	Mirror 0x00
0xE8	NOP	Mirror 0x00
0xE9	NOP	Mirror 0x00
0xEA	NOP	Mirror 0x00
0xEB	NOP	Mirror 0x00
0xEC	NOP	Mirror 0x00
0xED	NOP	Mirror 0x00
0xEE	NOP	Mirror 0x00
0xEF	NOP	Mirror 0x00
0xF0		
0xF1		
0xF2		
0xF3		
0xF4		
0xF5		
0xF6		

Continued on next page

Table A.1 – continued from previous page

Opcode	Command Name	Description
0xF7		
0xF8		
0xF9		
0xFA		
0xFB		
0xFC		
0xFD		
0xFE		
0xFF		

A.2 GP1 Command List

Table A.2: GP1 Commands (Top 2 Bits Ignored)

Opcode	Command Name	Description
0x00	RESET	Resets the GPU
0x01	RESET_CMD	Resets the command FIFO
0x02	ACK_IRQ	Acknowledge interrupt
0x03	DISP_EN	Enables display
0x04	DMA_DIR	Sets DMA direction
0x05	DISP_AREA_TL	Sets the display area's top-left corner
0x06	DISP_HRZ	Sets the display area's horizontal range
0x07	DISP_VTR	Sets the display area's vertical range
0x08	DISP_MODE	Sets various display settings

Continued on next page

Table A.2 – continued from previous page

Opcode	Command Name	Description
0x09	TX_DIS	Disables textures
0x0A	NOP	
0x0B	NOP	
0x0C	NOP	
0x0D	NOP	
0x0E	NOP	
0x0F	NOP	
0x10	GET_INFO	Gets GPU information based on argument
0x11	GET_INFO	Mirror 0x10
0x12	GET_INFO	Mirror 0x10
0x13	GET_INFO	Mirror 0x10
0x14	GET_INFO	Mirror 0x10
0x15	GET_INFO	Mirror 0x10
0x16	GET_INFO	Mirror 0x10
0x17	GET_INFO	Mirror 0x10
0x18	GET_INFO	Mirror 0x10
0x19	GET_INFO	Mirror 0x10
0x1A	GET_INFO	Mirror 0x10
0x1B	GET_INFO	Mirror 0x10
0x1C	GET_INFO	Mirror 0x10
0x1D	GET_INFO	Mirror 0x10
0x1E	GET_INFO	Mirror 0x10
0x1F	GET_INFO	Mirror 0x10
0x20	TX_DIS_ANC	Defunct texture disable command

Continued on next page

Table A.2 – continued from previous page

Opcode	**Command Name**	**Description**
0x21	NOP	
0x22	NOP	
0x23	NOP	
0x24	NOP	
0x25	NOP	
0x26	NOP	
0x27	NOP	
0x28	NOP	
0x29	NOP	
0x2A	NOP	
0x2B	NOP	
0x2C	NOP	
0x2D	NOP	
0x2E	NOP	
0x2F	NOP	
0x30	NOP	
0x31	NOP	
0x32	NOP	
0x33	NOP	
0x34	NOP	
0x35	NOP	
0x36	NOP	
0x37	NOP	
0x38	NOP	

Continued on next page

Table A.2 – continued from previous page

Opcode	**Command Name**	**Description**
0x39	NOP	
0x3A	NOP	
0x3B	NOP	
0x3C	NOP	
0x3D	NOP	
0x3E	NOP	
0x3F	NOP	

B CDROM Memory Map

B.1 CDROM Memory Map

Port	0		1		2		3	
Address	R	W	R	W	R	W	R	W
1F801800	Status Register							
1F801801	Response	Cmd	Response		Response		Response	AudioVol
1F801802	Data	Param	Data	IntEnable	Data	AudioVol	Data	AudioVol
1F801803	IntEnable	Request	IntFlag	IntFlag	IntEnable	AudioVol	IntFlag	ApplyVol

C Hardware Registers

Expansion Region 1

```
1F000000h 80000h Expansion Region (default 512 Kbytes, max 8 MBytes)
1F000000h 100h   Expansion ROM Header (IDs and Entrypoints)
```

Scratchpad

```
1F800000h 400h Scratchpad (1K Fast RAM) (Data Cache mapped to fixed address)
```

Memory Control 1

```
1F801000h 4    Expansion 1 Base Address (usually 1F000000h)
1F801004h 4    Expansion 2 Base Address (usually 1F802000h)
1F801008h 4    Expansion 1 Delay/Size (usually 0013243Fh; 512Kbytes 8bit-bus)
1F80100Ch 4    Expansion 3 Delay/Size (usually 00003022h; 1 byte)
1F801010h 4    BIOS ROM    Delay/Size (usually 0013243Fh; 512Kbytes 8bit-bus)
1F801014h 4    SPU_DELAY   Delay/Size (usually 200931E1h)
1F801018h 4    CDROM_DELAY Delay/Size (usually 00020843h or 00020943h)
1F80101Ch 4    Expansion 2 Delay/Size (usually 00070777h; 128-bytes 8bit-bus)
1F801020h 4    COM_DELAY / COMMON_DELAY (00031125h or 0000132Ch or 00001325h)
```

Peripheral I/O Ports

```
1F801040h 1/4  JOY_DATA Joypad/Memory Card Data (R/W)
1F801044h 4    JOY_STAT Joypad/Memory Card Status (R)
1F801048h 2    JOY_MODE Joypad/Memory Card Mode (R/W)
1F80104Ah 2    JOY_CTRL Joypad/Memory Card Control (R/W)
1F80104Eh 2    JOY_BAUD Joypad/Memory Card Baudrate (R/W)
1F801050h 1/4  SIO_DATA Serial Port Data (R/W)
1F801054h 4    SIO_STAT Serial Port Status (R)
1F801058h 2    SIO_MODE Serial Port Mode (R/W)
1F80105Ah 2    SIO_CTRL Serial Port Control (R/W)
1F80105Ch 2    SIO_MISC Serial Port Internal Register (R/W)
1F80105Eh 2    SIO_BAUD Serial Port Baudrate (R/W)
```

Memory Control 2

```
1F801060h 4/2  RAM_SIZE (usually 00000B88h; 2MB RAM mirrored in first 8MB)
```

Interrupt Control

```
1F801070h 2    I_STAT - Interrupt status register
1F801074h 2    I_MASK - Interrupt mask register
```

DMA Registers

```
1F80108xh      DMA0 channel 0 - MDECin
1F80109xh      DMA1 channel 1 - MDECout
1F8010Axh      DMA2 channel 2 - GPU (lists + image data)
1F8010Bxh      DMA3 channel 3 - CDROM
1F8010Cxh      DMA4 channel 4 - SPU
1F8010Dxh      DMA5 channel 5 - PIO (=Expansion Port?)
1F8010Exh      DMA6 channel 6 - OTC (reverse clear OT) (GPU related)
1F8010F0h      DPCR - DMA Control register
1F8010F4h      DICR - DMA Interrupt register
1F8010F8h      unknown
1F8010FCh      unknown
```

Timers (aka Root counters)

1F80110xh	Timer 0 Dotclock
1F80111xh	Timer 1 Horizontal Retrace
1F80112xh	Timer 2 1/8 system clock

CDROM Registers (Address.Read/Write.Index)

1F801800h.x.x	1	CD Index/Status Register (Bit0-1 R/W, Bit2-7 Read Only)
1F801801h.R.x	1	CD Response Fifo (R) (usually with Index1)
1F801802h.R.x	1/2	CD Data Fifo - 8bit/16bit (R) (usually with Index0..1)
1F801803h.R.0	1	CD Interrupt Enable Register (R)
1F801803h.R.1	1	CD Interrupt Flag Register (R/W)
1F801803h.R.2	1	CD Interrupt Enable Register (R) (Mirror)
1F801803h.R.3	1	CD Interrupt Flag Register (R/W) (Mirror)
1F801801h.W.0	1	CD Command Register (W)
1F801802h.W.0	1	CD Parameter Fifo (W)
1F801803h.W.0	1	CD Request Register (W)
1F801801h.W.1	1	Unknown/unused
1F801802h.W.1	1	CD Interrupt Enable Register (W)
1F801803h.W.1	1	CD Interrupt Flag Register (R/W)
1F801801h.W.2	1	Unknown/unused
1F801802h.W.2	1	CD Audio Volume for Left-CD-Out to Left-SPU-Input (W)
1F801803h.W.2	1	CD Audio Volume for Left-CD-Out to Right-SPU-Input (W)
1F801801h.W.3	1	CD Audio Volume for Right-CD-Out to Right-SPU-Input (W)
1F801802h.W.3	1	CD Audio Volume for Right-CD-Out to Left-SPU-Input (W)
1F801803h.W.3	1	CD Audio Volume Apply Changes (by writing bit5=1)

GPU Registers

1F801810h.Write	4	GP0 Send GP0 Commands/Packets (Rendering and VRAM Access)
1F801814h.Write	4	GP1 Send GP1 Commands (Display Control)
1F801810h.Read	4	GPUREAD Read responses to GP0(C0h) and GP1(10h) commands
1F801814h.Read	4	GPUSTAT Read GPU Status Register

MDEC Registers

1F801820h.Write	4	MDEC Command/Parameter Register (W)
1F801820h.Read	4	MDEC Data/Response Register (R)
1F801824h.Write	4	MDEC Control/Reset Register (W)
1F801824h.Read	4	MDEC Status Register (R)

SPU Voice 0..23 Registers

1F801C00h+N*10h	4	Voice 0..23 Volume Left/Right
1F801C04h+N*10h	2	Voice 0..23 ADPCM Sample Rate
1F801C06h+N*10h	2	Voice 0..23 ADPCM Start Address
1F801C08h+N*10h	4	Voice 0..23 ADSR Attack/Decay/Sustain/Release
1F801C0Ch+N*10h	2	Voice 0..23 ADSR Current Volume
1F801C0Eh+N*10h	2	Voice 0..23 ADPCM Repeat Address

SPU Control Registers

1F801D80h	4	Main Volume Left/Right
1F801D84h	4	Reverb Output Volume Left/Right
1F801D88h	4	Voice 0..23 Key ON (Start Attack/Decay/Sustain)
1F801D8Ch	4	Voice 0..23 Key OFF (Start Release)
1F801D90h	4	Voice 0..23 Channel FM (pitch lfo) mode
1F801D94h	4	Voice 0..23 Channel Noise mode
1F801D98h	4	Voice 0..23 Channel Reverb mode
1F801D9Ch	4	Voice 0..23 Channel ON/OFF (status)
1F801DA0h	2	Unknown? (R) or (W)
1F801DA2h	2	Sound RAM Reverb Work Area Start Address
1F801DA4h	2	Sound RAM IRQ Address
1F801DA6h	2	Sound RAM Data Transfer Address
1F801DA8h	2	Sound RAM Data Transfer Fifo
1F801DAAh	2	SPU Control Register (SPUCNT)
1F801DACh	2	Sound RAM Data Transfer Control
1F801DAEh	2	SPU Status Register (SPUSTAT)
1F801DB0h	4	CD Volume Left/Right
1F801DB4h	4	Extern Volume Left/Right
1F801DB8h	4	Current Main Volume Left/Right
1F801DBCh	4	Unknown? (R/W)

SPU Reverb Configuration Area

1F801DC0h	2	dAPF1	Reverb APF Offset 1
1F801DC2h	2	dAPF2	Reverb APF Offset 2
1F801DC4h	2	vIIR	Reverb Reflection Volume 1
1F801DC6h	2	vCOMB1	Reverb Comb Volume 1
1F801DC8h	2	vCOMB2	Reverb Comb Volume 2
1F801DCAh	2	vCOMB3	Reverb Comb Volume 3
1F801DCCh	2	vCOMB4	Reverb Comb Volume 4
1F801DCEh	2	vWALL	Reverb Reflection Volume 2
1F801DD0h	2	vAPF1	Reverb APF Volume 1
1F801DD2h	2	vAPF2	Reverb APF Volume 2
1F801DD4h	4	mSAME	Reverb Same Side Reflection Address 1 Left/Right
1F801DD8h	4	mCOMB1	Reverb Comb Address 1 Left/Right
1F801DDCh	4	mCOMB2	Reverb Comb Address 2 Left/Right
1F801DE0h	4	dSAME	Reverb Same Side Reflection Address 2 Left/Right
1F801DE4h	4	mDIFF	Reverb Different Side Reflection Address 1 Left/Right
1F801DE8h	4	mCOMB3	Reverb Comb Address 3 Left/Right
1F801DECh	4	mCOMB4	Reverb Comb Address 4 Left/Right
1F801DF0h	4	dDIFF	Reverb Different Side Reflection Address 2 Left/Right
1F801DF4h	4	mAPF1	Reverb APF Address 1 Left/Right
1F801DF8h	4	mAPF2	Reverb APF Address 2 Left/Right
1F801DFCh	4	vIN	Reverb Input Volume Left/Right

SPU Internal Registers

```
  1F801E00h+N*04h  4 Voice 0..23 Current Volume Left/Right
  1F801E60h      20h Unknown? (R/W)
  1F801E80h     180h Unknown? (Read: FFh-filled) (Unused or Write only?)
```

Expansion Region 2 (default 128 bytes, max 8 KBytes)

```
  1F802000h      80h Expansion Region (8bit data bus, crashes on 16bit access?)
```

Expansion Region 2 - Dual Serial Port (for TTY Debug Terminal)

```
  1F802020h/1st    DUART Mode Register 1.A (R/W)
  1F802020h/2nd    DUART Mode Register 2.A (R/W)
  1F802021h/Read   DUART Status Register A (R)
  1F802021h/Write  DUART Clock Select Register A (W)
  1F802022h/Read   DUART Toggle Baud Rate Generator Test Mode (Read=Strobe)
  1F802022h/Write  DUART Command Register A (W)
  1F802023h/Read   DUART Rx Holding Register A (FIFO) (R)
  1F802023h/Write  DUART Tx Holding Register A (W)
  1F802024h/Read   DUART Input Port Change Register (R)
  1F802024h/Write  DUART Aux. Control Register (W)
  1F802025h/Read   DUART Interrupt Status Register (R)
  1F802025h/Write  DUART Interrupt Mask Register (W)
  1F802026h/Read   DUART Counter/Timer Current Value, Upper/Bit15-8 (R)
  1F802026h/Write  DUART Counter/Timer Reload Value,  Upper/Bit15-8 (W)
  1F802027h/Read   DUART Counter/Timer Current Value, Lower/Bit7-0 (R)
  1F802027h/Write  DUART Counter/Timer Reload Value,  Lower/Bit7-0 (W)
  1F802028h/1st    DUART Mode Register 1.B (R/W)
  1F802028h/2nd    DUART Mode Register 2.B (R/W)
  1F802029h/Read   DUART Status Register B (R)
  1F802029h/Write  DUART Clock Select Register B (W)
  1F80202Ah/Read   DUART Toggle 1X/16X Test Mode (Read=Strobe)
  1F80202Ah/Write  DUART Command Register B (W)
  1F80202Bh/Read   DUART Rx Holding Register B (FIFO) (R)
  1F80202Bh/Write  DUART Tx Holding Register B (W)
  1F80202Ch/None   DUART Reserved Register (neither R nor W)
  1F80202Dh/Read   DUART Input Port (R)
  1F80202Dh/Write  DUART Output Port Configuration Register (W)
  1F80202Eh/Read   DUART Start Counter Command (Read=Strobe)
  1F80202Eh/Write  DUART Set Output Port Bits Command (Set means Out=LOW)
  1F80202Fh/Read   DUART Stop Counter Command (Read=Strobe)
  1F80202Fh/Write  DUART Reset Output Port Bits Command (Reset means Out=HIGH)
```

Expansion Region 2 - Int/Dip/Post

```
  1F802030h Secondary IRQ10 Flags
  1F802040h "Dip switches"
  1F802041h POST (external 7 segment display to indicate BIOS boot status)
  1F802070h POST2 (similar to POST, but PS2 BIOS uses this address)
```

Expansion Region 2 - Nocash Emulation Expansion

```
  1F802060h Emu-Expansion ID1 "E" (R)
  1F802061h Emu-Expansion ID2 "X" (R)
  1F802062h Emu-Expansion ID3 "P" (R)
  1F802063h Emu-Expansion Version (01h) (R)
  1F802064h Emu-Expansion Enable1 "O" (R/W)
  1F802065h Emu-Expansion Enable2 "N" (R/W)
  1F802066h Emu-Expansion Halt (R)
  1F802067h Emu-Expansion Turbo Mode Flags (R/W)
```

Expansion Region 3 (default 1 byte, max 2 MBytes)

```
  1FA00000h - Not used by BIOS or any PSX games
  1FA00000h - POST3 (similar to POST, but PS2 BIOS uses this address)
```

BIOS Region (default 512 Kbytes, max 4 MBytes)

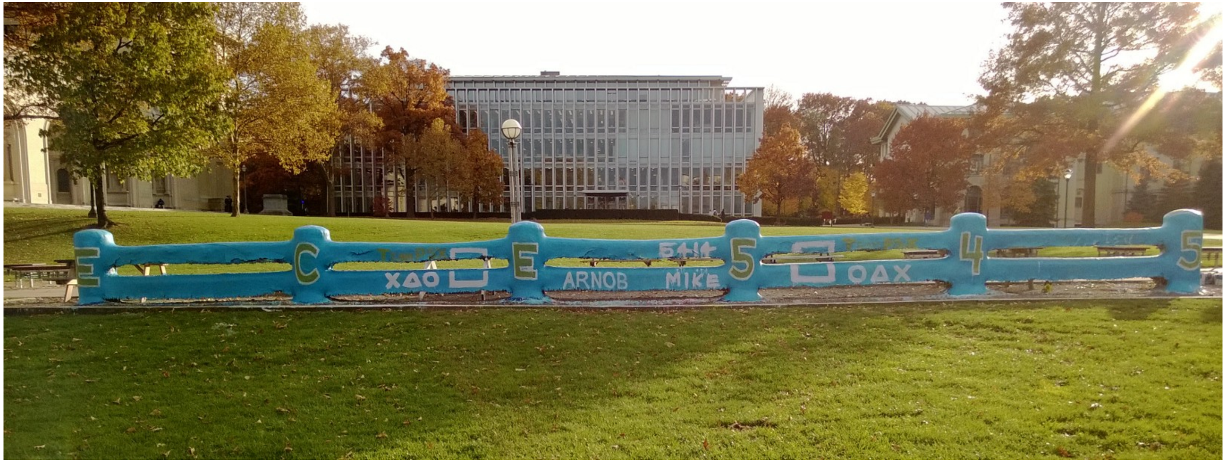
```
  1FC00000h 80000h   BIOS ROM (512Kbytes) (Reset Entrypoint at BFC00000h)
```

Memory Control 3 (Cache Control)

```
  FFFE0130h 4        Cache Control
```

D The Fence

Figure D.1: Front of the Fence



They texted me at 3 AM. I ignored them. -Anita

Figure D.2: Back of the Fence