

# Team Genesis Final Report



Alex Etling, Ben Joyce, Kun Li

18-545

Fall 2012

# Table of Contents

<b>OVERVIEW .....</b>	<b>1</b>
<b>SEGA GENESIS .....</b>	<b>3</b>
<b>COMPONENT DESCRIPTION .....</b>	<b>4</b>
<b>BLOCK DIAGRAM/MEMORY MAP .....</b>	<b>7</b>
<b>APPROACH .....</b>	<b>9</b>
<b>PORTING .....</b>	<b>10</b>
<b>AUDIO .....</b>	<b>14</b>
<b>HARDWARE .....</b>	<b>20</b>
<b>TESTING AND VERIFICATION .....</b>	<b>25</b>
<b>WORDS OF WISDOM/CONCLUSIONS .....</b>	<b>27</b>
<b>INDIVIDUAL PAGES.....</b>	<b>28</b>

## OVERVIEW

Our team created a fully functional Sega Genesis console on an FPGA, including video, sound, cartridge, and controller interfaces. Our system looks and plays almost exactly like the original console, and is capable of running any game (as far as we tested). Two players are supported, and the user can toggle between the cartridge reader and a game ROM stored in on-board Flash memory, allowing them to play games they do not have a physical copy of.

The Sega Genesis system is composed of five main components: two CPUs, a video processor, and two sound processors. In addition, there are three main memory modules, 64 KB of work RAM, 8 KB of sound RAM, and 64 KB of video RAM, and many assorted registers for each chip. The entire system is built around two busses and two memory-mapped address spaces, one each for the two CPUs. A bus arbiter allows either CPU to access a module on the other's bus by requesting that bus for itself. The system peripherals, including the cartridge and controllers, are also accessed through the bus/address space configuration. A Direct Memory Access (DMA) controller can also take control of the bus to enable fast transfer of data from either the game ROM or the work RAM to the video processor.

For hardware, we used a standard 3-button Sega Genesis controller for Player 1 and a keyboard for Player 2. Video is output through the DVI port on the board to the VGA input of a monitor through a DVI-to-VGA converter. Audio is played through the AC'97 sound driver and output through the Line Out port to standard headphones or speakers. The game cartridge and Player 1 controller are connected to the peripheral I/O pins on the FPGA board, while the keyboard is connected through the PS/2 input port.

In this paper, we describe the Sega Genesis system as a whole, the internal details of each chip, and the hardware peripherals. We also discuss the challenges we faced to get our system working, as well as lessons learned along the way.



Figure 1. Final version of the Genesis on Demo Day

## **SEGA GENESIS**

The Sega Genesis, also known as the Sega Mega Drive, was first released in Japan in 1988, in the U.S. in 1989, and in Europe in 1990. It was released 2 years before the Super Nintendo Entertainment System, and was the first Sega product to garner any sort of support in the United States. It initially dominated the 8-bit NES system, but the competition between the Genesis and the SNES would come to define the 16-bit console era. Throughout this period the Genesis was able to keep a majority share of the market share in the US, with popular games and catchy slogans like “Genesis does what Nintendon’t”.

Genesis games were known for many different things. They were some of the first “sponsored” games, with Pat Riley Basketball and Joe Montana Football. On top of this, Sega’s Mortal Kombat was so violent they had to create a Video Game Rating Council, the first of its kind. This would go on as the framework for the general rating system of all games. The Genesis also had the first major appearance of Sonic the Hedgehog, one of Sega’s most well known characters.

The Genesis contained 5 chips to make all of its operations work. The main processor chip was the Motorola 68000 (68k) 16-bit processor, which runs at 7.67 MHZ. Its coprocessor is the Zilog Z80 8-bit processor, which runs at 3.58 MHZ. The visual display is done through a Yamaha YM7101 video processor. There are 2 sound chips, the Yamaha YM2612 and the Texas Instrument SN76489.

The main operation of the Genesis works through interaction with the cartridge. The cartridge itself is just a ROM with a max of 4 MB of data. The processor steps through cartridge instructions just like a normal CPU, and based on these decides to transfer data to or write registers in the other modules. Input from the controllers allows the user to manipulate the game execution flow.

## COMPONENT DESCRIPTION

### *Motorola 68000:*

This is the main CPU of the system. Its primary task is to read and execute instructions from the game ROM, then forward data or write to registers in the Z80, video processor, or sound chips. The 68k has a memory mapped address space with access to the game ROM, a 64 KB work RAM, video processor, I/O pins, and the entire Z80 address space. It basically functioned as the task manager of the system; it made sure that each other processor had the information they needed to operate and executed its own instructions, communicated with the peripheral I/O devices, and executed game code.

The 68k is a CISC processor designed by Motorola (now Freescale) and introduced in 1979. It has a 24-bit external address, eight 32-bit general purpose registers, and 56 instructions with a minimum size of 16 bits. The processor uses a R/ $\bar{W}$  line, upper and lower data strobes to specify valid bytes on its data line, and a data acknowledge line to handshake with any peripheral module. There are also 3-bits of interrupt control, allowing for 6 levels of external interrupts (the Sega Genesis uses 3 of these).

### *Zilog Z80:*

The Z80 is the coprocessor of the system. This chip mainly dealt with controlling the two sound chips. In most games, the Z80 looped through a small portion of driver code stored in the 8 KB sound RAM and fed data to the sound chips to play music. It has its own memory mapped address that could access either sound chip, the sound RAM, or the entire 68k address space through a bank switching mechanism. Since all elements of the system are shared, either processor needs to request the other processor's bus in order to access modules that are not directly addressed in its own address space. For example, the Z80 can access game ROM only by requesting the bus and using the appropriate bank to access the ROM portion of the 68k address space. The Z80 is an 8-bit processor designed by Zilog and sold from 1976 onwards. The Z80 dominated the microcomputer market from the mid 70's to the mid 80's. Its programming set and registers are very

similar to the ones that are found in x86. It uses a dual register set, has 252 different opcodes, and 4 opcode prefixes.

***Yamaha YM7101:***

This is the video processor (VDP) responsible for translating game data into images seen on the screen. The VDP is capable of rendering two background layers and one sprite layer per frame, with adjustable priorities between layers. Up to 32 simultaneous colors are available using color palettes stored in a 64x9-bit color RAM. This includes 16 colors for sprites and background and 16 for background only. The VDP supports resolutions of 256x192 and 256x224 pixels, 8x8 or 8x16 characters and sprites, as well as horizontal, vertical, and partial screen scrolling. A 64 KB video RAM (VRAM) stores pattern data, sprite tables, and other assorted information needed to render the scene. The VDP also has a 40x10-bit vertical scroll RAM and 23 8-bit registers. Finally, a Direct Memory Access (DMA) engine is used to quickly transfer data from either the game ROM or work RAM to the VDP.

***Yamaha Y2612:***

This is a 6 channel FM synthesizer chip used to produce the main background music and sound effects in games. Each channel uses four “operators” combined in different ways to produce notes in different instrument voices. An adjustable attack-decay-sustain-release envelope controls the attack and duration of notes. There are also two interval timers, a low frequency oscillator, and an undocumented SSG-EG mode. An optional 8-bit PCM stream can replace channel 6 to play raw audio data directly from the game ROM. The YM2612 is completely controlled by writing to 213 different 8-bit registers. The chip performs internal calculations using sine and power lookup tables to generate frequency modulated notes. The output is 14-bit signed PCM data generated at 53 kHz.

***Texas Instrument SN76489:***

This is a Programmable Sound Generator (PSG) capable of producing 3 channels of square wave tones and 1 channel of noise. Like the YM2612, the PSG is controlled by writing to different registers to control the frequency and attenuation of each channel.

The noise channel is capable of producing either white or periodic noise. The output of this chip is an 11-bit signed value which is summed with the output from the YM2612 to produce the final audio data. The PSG was typically used to produce simple notes, sound effects, and noise which the YM2612 may not be capable of synthesizing.



## BLOCK DIAGRAM / MEMORY MAP

A block diagram of our implementation of the Sega Genesis is shown below (Fig. 2). This configuration is based off of the original Sega Genesis and modified to work with various features of the Xilinx FPGA board we are working with. The 68k is the main processor and communicates one at a time with various peripheral processors and I/O devices through a memory mapped address space (Fig. 3). In our implementation, the address space is treated as enable lines for reading and writing data to various modules. A top level module containing control logic to mimic the busses and address spaces connects all of the pieces together and deals with timing management.

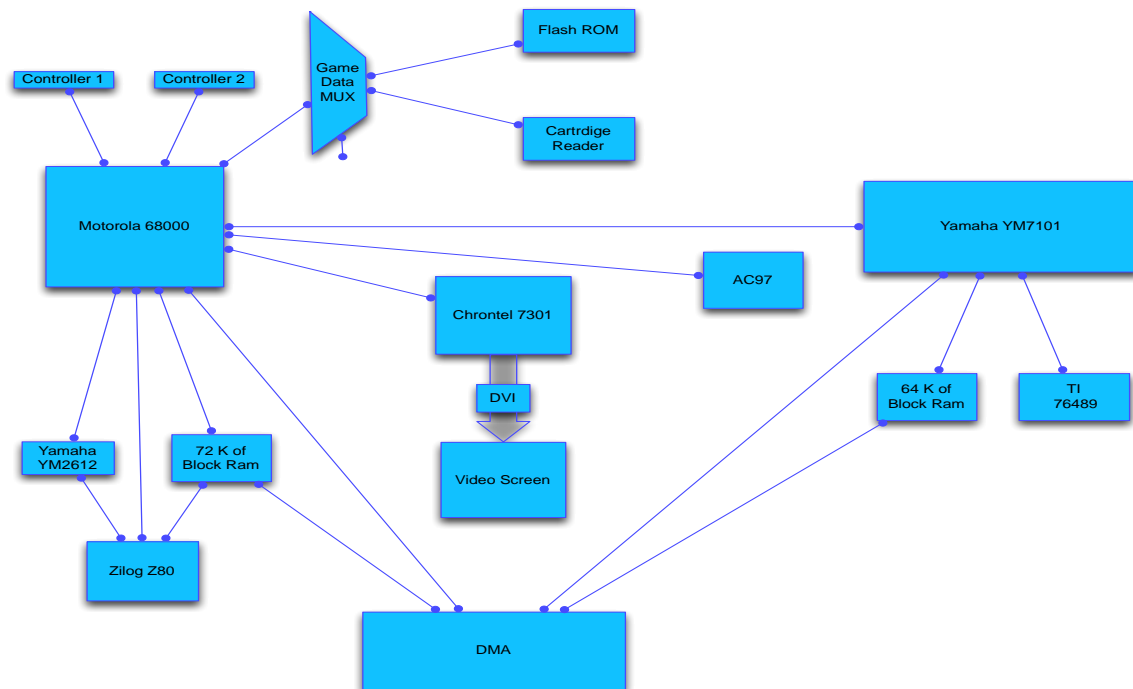


Figure 2. Block diagram of our Sega Genesis implementation. The 68k is the central communication hub for data, commands, and peripheral I/O devices.

The 68k uses a 24-bit memory mapped address space to communicate with the peripheral modules, while the Z80 uses a 16-bit address space (Fig. 3). The 68k can access the entirety of the Z80 address space by using the lower 16-bits with the upper 8-bits set to 0xA0. Similarly, the Z80 can also access the entire 68k address space through a bank switching mechanism and bus arbiter. This gives each CPU access to any module in the system, although only one of them can access a given module at a time.

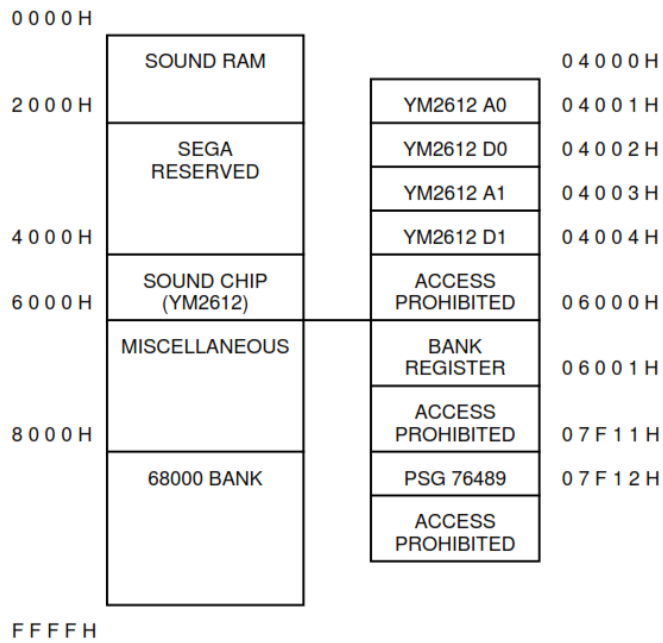
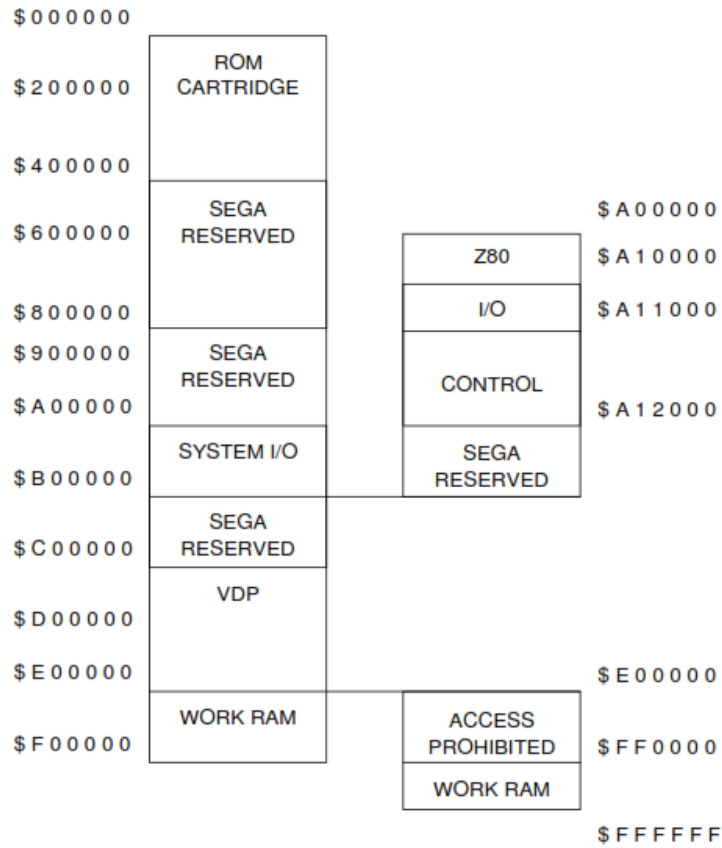


Figure 3. Memory map of Sega Genesis. Top shows address space of 68k processor, bottom shows address space of Z80.

## APPROACH

We used the fpgagen project by Gregory Estrade ([code.google.com/p/fpgagen/](https://code.google.com/p/fpgagen/)) as the starting point for our project. This is an open source project which took OpenCores versions of the 68k and Z80 processors and integrated them with a self-written video processor and glue logic to create a Sega Genesis system on an Altera DE1 board. This project managed to get video mostly working for some games, but with no sound or peripheral hardware. We wanted to port the project from the Altera board to our Xilinx board, add sound and hardware peripherals, and fix bugs in the video output to get a complete system working.

This design decision was made with respect to historical data from previous iterations of this class. Most groups that attempted a video game console had been unable to complete their system in one semester, even when starting off with an open source CPU. Even those who got the system working ended up spending all of their time working on the video processor and integration, and either omitted sound or used an open source or incomplete implementation. While technically the game will play without sound, anybody who grew up playing these consoles knows that the experience is not complete without the classic video game music. In addition, the Sega Genesis is on the upper end in terms of system complexity, and the only comparable system attempted in this class, the SNES, had not been completed. We believe that our design decision gave us the best chance to get the system working, while providing sufficient complexity to keep us occupied. We also felt that using open source code is a legitimate practice in a real-world setting; we saw no value in reinventing the wheel while there were other novel challenges to be faced. In the following sections, we will go into more details about the challenges faced in building off the fpgagen project, as well as our original contributions.

## **PORTING**

The porting process required mapping all of the peripheral modules on the Altera DE1 board used by the fpgagen project onto functionally equivalent modules on the Xilinx XUPV5-LX110T board. A brief description of each Altera module is listed below along with the steps taken to port it over and the associated challenges:

### ***SDRAM:***

The DE1 board used an SDRAM chip to mimic the combined work RAM and sound RAM used by both processors on the Sega Genesis. The closest functional module on the Xilinx board was a DDR2 chip. However, DDR2 is somewhat tricky to access, and would require either using a Xilinx Memory Interface Generator IP core with FIFO inputs/outputs, or writing and debugging our own controller. We simplified the problem by taking advantage of our FPGA's high logic cell count and the relatively small amount of memory (by today's standard) required by the Genesis and using a dedicated 72KB block RAM as the combined work/sound RAM. The challenges in porting this memory module over were in understanding the SDRAM controller well enough to modify it and going from a bidirectional data bus with separate read and write commands to dual port data lines with a single  $R/\bar{W}$  line. We had to change the control logic, use high impedance states, and adjust the timing to get this to work.

### ***SRAM:***

The DE1 board used SRAM as the video RAM accessed by the VDP. The Xilinx board also has an SRAM chip, but it shares address and data lines with the Flash chip we also need. In lieu of arbitrating between the two chips and dealing with any issues that might arise such as timing, simultaneous access, or inappropriate data floating on the shared lines, we again used a 64KB block RAM module here. The same challenges associated with the SDRAM held true here.

### ***Flash:***

The game ROM was stored on a Flash memory chip on the DE1 board. The Xilinx board also had a Flash chip which we could program using iMPACT, the

command line tool *xxd*, Xilinx's *promgen* command, and a Sega Genesis game ROM binary. The issues associated with this chip were matching the new timing constraints for reads and dealing with a 16-bit bus vs. an 8-bit bus on the Altera board. The Flash chip on the Xilinx board is also little endian, so we had to flip the read order after debugging this in ChipScope.

### ***PLL:***

The DE1 board used a PLL to multiply a 27 MHz clock to get a 54 MHz clock, which was then used as a global clock and divided by 7 to get the 68000 processor clock. We at first tried using the 27 MHz clock and the PLL on the Xilinx board to the same effect, but after instantiating the PLL we found that its minimum output frequency was 400 Hz, too high for our purposes. We instead used a 100 MHz clock and divided it down by 13 to get the processor clock. However, this faster global clock caused some timing issues on our board, where signals were not propagating to the appropriate registers in time before the next clock tick. This caused an insidious bug where every time we would change something in our code, the video output would be completely different after synthesizing. We believed that the VDP itself had errors, leading to a fair amount of time spent identifying and tracking down this bug. We eventually got a Xilinx Digital Clock Manager (DCM) primitive working to generate a 54 MHz clock, which was able to meet the board timing constraints.

### ***ALTSYNCRAM:***

The DE1 board used a block RAM equivalent to store the color palettes and background tables. This was easily converted into block RAM on the Xilinx board.

### ***VGA:***

The fpgagen project generated VGA data and sent it directly to the VGA port for output. On our Xilinx board, we only had a DVI port available for video output. Also, instead of directly outputting our data to the pins we want, we must first send our data through a Chrontel 7301 DVI transmitter device, which can then do some processing on the data and output it on the DVI port. We configured the CH7301 through I<sup>2</sup>C to send

analog data through the DVI port. From the fpgagen project, the video processor outputs 12 bit color, but since our board supports 24 bit color we mapped the pixel data to 8 bits per color instead of 4 by multiplying by 16. While the fpgagen project pixel data and sync signals could be output directly to the VGA pins, we had to first go through a chip which requires very specific timing for horizontal sync, vertical sync, and data enable signals. We ran into problems because if the sync signals are not correct, the CH7301 will not output any data, making it hard to pinpoint the problem. We were able to get output from the Xilinx board by generating these signals by ourselves, but we still need to use the sync signals from the fpgagen project to get the video displayed on the screen correctly.

### ***Video:***

After getting the DVI controller working properly, we received very garbled output from the VDP using games that had previously been tested on the fpgagen project. We were able to find an emulator that would let us look at the VRAM and color RAM so we could use it to find where ours was incorrect. This took several weeks and involved tracing through every line of the original code. Once we got the output to look mostly correct, there were still many unimplemented or buggy features.

First, many of the sprites were looking fuzzy. This problem was because whenever we wrote a value to VRAM, that value would immediately appear on the data out lines and cause our VDP to read incorrect data. Xilinx allowed us to change the configuration of VRAM so we could write to memory but still hold the data on the output lines constant.

The VDP would sometimes display sprites or backgrounds on the edges of the screen when there was supposed to be something else, usually a solid color, there instead. This mostly happened when a game scrolled vertically. This is because there was a mode register that was being set in the VDP, but was being ignored. One of the modes was to display a solid background color instead of any sprites or background. Once we implemented the different VDP modes, the problem was fixed. There was also a register that enabled the display, but it was sometimes being set in the middle of a scan line so the

display would start in the middle of a screen. This was fixed by only latching this register at the beginning of each line.

Also, the original VDP did not implement shadows or highlighting, so we were able to find in the documentation how to implement that to make the video output look more realistic.

## AUDIO

### YM2612:

The YM2612 is a six-channel FM synthesizer which produces 14-bit signed audio output at a rate of 53 kHz. Each channel is capable of playing a single note in one of many different instrument voices. A channel is composed of four operators, where each operator represents a single sine wave that can be modulated in amplitude and frequency (Fig. 4). Operators have an input signal  $I(t)$ , a phase generator which gives the frequency  $\omega_c$ , and an envelope generator which gives the amplitude  $A(t)$ . The input signal is either the output of another operator, the output of the operator fed back to itself, or nothing.

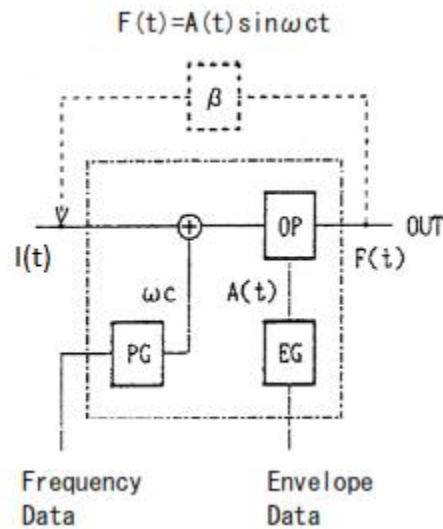


Figure 4. Block diagram of single operator.

If the input signal is from another operator with frequency  $\omega_m$ , the output can be given by:

$$F(t) = A(t)\sin(\omega_c t + I(t) \sin(\omega_m t)) \quad (1)$$

Where the input operator is said to *modulate* the frequency of the *carrier* operator. Similarly, if the operator uses self-feedback, the output is given by the equation:



$$F(t) = A(t)\sin(\omega_c t + \beta F(t)) \quad (2)$$

Where  $\beta$  is a feedback factor. In the simple case where there is no input, the operator is a simple amplitude-modulated sine wave given by:

$$F(t) = A(t)\sin(\omega_c t) \quad (3)$$

The four operators can be combined using up to 8 different “algorithms” (Fig. 5). Each algorithm has designated modulator and carrier (shaded) operators, as shown below. The carrier operators are summed together to produce the final output. The different algorithms produce different characteristic voices, as described in the table.

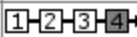
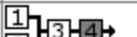



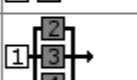
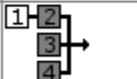
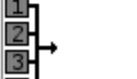
Algorithm #	Layout	Suggested uses
0		Distortion guitar, “high hat chopper” (?) bass
1		Harp, PSG (programmable sound generator) sound
2		Bass, electric guitar, brass, piano, woods
3		Strings, folk guitar, chimes
4		Flute, bells, chorus, bass drum, snare drum, tom-tom
5		Brass, organ
6		Xylophone, tom-tom, organ, vibraphone, snare drum, base drum
7		Pipe organ

Figure 5. Algorithms used to combine operators in a single channel.

The phase generator (PG) is implemented as a simple counter that increases by a set amount on each clock tick (1.28 MHz). This value determines the frequency of the note and is calculated from four parameters: frequency, octave, detune, and multiple. In a single channel, all four operators typically have the same frequency and octave, but can

vary in detune and multiple to produce different characteristics of sound. Channels 3 and 6 have a special mode, which allows the frequency and octave of each operator to be set independently when enabled. This is used, for example, in Streets of Rage to produce the punching sounds.

The envelope generator (EG) produces an Attack-Decay-Sustain-Release (ADSR) envelope to mimic the effects of starting, stopping, and sustaining a note. Figure 6 summarizes the four stages. The y axis represents attenuation, up to a maximum of -96 dB. At  $t = 0$ , the note is fully attenuated and Key On is triggered. The EG enters the Attack phase, where the attenuation increases exponentially towards the Total Level (TL). The exponential rate is given by the Attack Rate (AR). Once the attenuation reaches TL, the EG moves into the Decay phase, where the attenuation decreases linearly at the decay rate D1R. When it hits the sustain level, it decreases linearly at the sustain rate D2R. During any of the other three phases, Key Off can be triggered, at which point the EG immediately goes into the Release stage. Again, the attenuation declines at a linear rate given by the Release Rate (RR), until it reaches full attenuation. The output of the EG thus determines the amplitude of the sine wave at any given time.

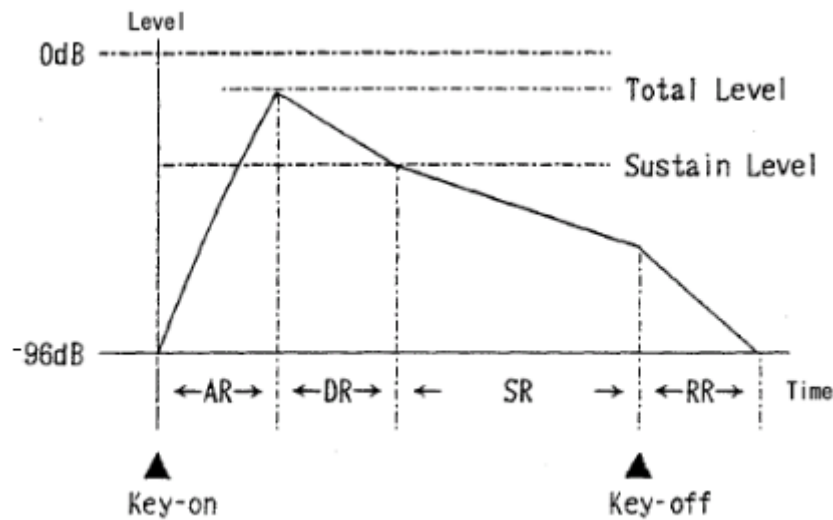


Figure 6. Attack-Decay-Sustain-Release envelope.

Finally, the channel calculations are performed according to equations (1)-(3). The sine value is calculated from the phase by a lookup table to avoid having to do

trigonometric calculations. In our implementation, as in the original YM2612 chip, the LUT values are in the logarithmic scale so that multiplication by the attenuation (which is already in logarithmic scale) becomes a simple addition. Only a quarter of the period is stored to preserve space. Finally, the calculated value is converted from a logarithmic attenuation to a linear amplitude using a power LUT.

The YM2612 is controlled by a wide range of registers, which determine all of the parameters of the above calculations, such as algorithm, frequency, attack rate, total level, etc. These registers can be written to by either the 68k or Z80 processor. The processor first writes the 8-bit address of the target register, followed by the 8-bit value stored in the register.

To build the YM2612 core, we first figured out the physical implementation of the above algorithm, including bit widths and timing considerations, by extensively reading research presented on the SpritesMind.net forums and the MAME emulator source code. We worked out a sample calculation by hand for a test note using the register configuration presented in the Sega Genesis Software Manual, which produces a grand piano sound when played. We used a C program to generate the binary LUTs for the sine and power values, and hard coded them into my VHDL file. We then implemented the PG, EG, and channel calculations and simulated them with the test registers in Xilinx iSim. Once this was consistent with our hand calculations, we built a basic top module to connect the FM chip with the AC'97 driver code. Unfortunately, playing this note only resulted in a short 'ping' sound from the speakers. At this point, we were not sure if there was a problem with my FM module or the AC'97 driver. Switching the byte order (in case the AC'97 codec is little endian) produced a longer fuzzy note, but it still did not sound correct. We decided to double check my output against the MAME core to see which part of the system was at fault. We hacked into the sound driver code, built a main program around it, and used printf statements to debug the output at different stages. Through this, we managed to fix a few small bugs in my code, and eventually saw that our output was basically cycle accurate with the MAME core. To check the expected output of these computations, we wrote the binary data to a file and played it back through Audacity. The result was the expected piano note. However, my initial impression was that we were supposed to sample the output every 24

cycles. If we did this, we got the short ‘ping’ sound that we had heard on the board. By slowing everything down by 24 cycles on the FPGA, we were able to re-create the piano note.

We were confused about this apparent discrepancy between the documentation and the results of our testing. We tried to build a FIFO buffer to convert to the slower 48 kHz sample rate used by the AC’97, but this would inevitably overflow, even if we clear out the buffer every time there is silence. After asking for help on the SpritesMind forums, we realized we had made a mistake in my understanding of the clock used by the FM chip. In fact, our calculations should have been performed 24x slower. With this knowledge, we re-routed the test program to the rest of the Genesis system and ran a test “song player” game file through the Genesis writing to the YM2612 core to produce notes through the AC’97 sound driver. This managed to produce Beethoven’s “Ode to Joy” using a single channel.

We then moved on to implementing the other five channels. We had written our code with loop unrolling in mind, so we simply changed the parameters of the loops to attain the other channels. However, Xilinx would crash if we tried to synthesize all six channels, so we settled for three. We managed to get some music playing from Gunstar Heroes and Sonic Spinball and used this to debug errors in the sound output. To solve the synthesis issue, we checked the Xilinx forums for a solution and found that ISE might be running out of memory. Using the *top* command in Linux, we found that this was indeed the case, but that Xilinx was using up all 16 GB of RAM on the lab machines! This was either due to a memory leak or inefficiencies in our code. We decided to modularize the code for a single channel instead of using loop unrolling, and this allowed the entire project to synthesize successfully. We spent the last week checking audio output against the original games to identify any remaining errors in our implementation.

### ***SN76489:***

The SN76489, or Programmable Sound Generator (PSG), is a four channel tone and noise generator. There are three channels of tones, or square waves with adjustable frequency and magnitude. There is also a noise channel, which can be toggled between white or periodic noise, also with adjustable frequency and magnitude. Like the

YM2612, the PSG is controlled from either CPU by writing different commands to the corresponding memory mapped address. The amplitudes are set by a LUT to form an 11-bit output which is directly summed with the output from the YM2612. The noise channels are implemented as a linear-feedback shift register. After implementing the YM2612, the PSG was fairly straightforward to implement and integrate to the rest of the system. We used switches to isolate the different channels of sound and confirm that everything was working.

### *AC'97*

We output our audio data through the AC'97 audio driver located on the Xilinx board. We learned how to use this module during Lab 2, and transferred our implementation directly to this system. We sampled the output as 16-bit PCM data at 48 kHz. Since the samples were generated at 53 kHz, there was minimal loss of data despite the unsynchronized sampling rates. To obtain the 16-bit data, we shifted our 14-bit output left two bits and filled in the lower 2 bits with the sign bit.

## **HARDWARE**

### ***Controller / Interface:***

The controller has 8 buttons on it: up, down, left, right, start, a, b, and c. The pin out for the controller is a standard db9 connector. The signals for the controller are multiplexed as shown:

Up Signal = Pin 1 (with Select HIGH or LOW).

Down Signal = Pin 2 (with Select HIGH or LOW).

Left Signal = Pin 3 (with Select HIGH).

Right Signal = Pin 4 (with Select HIGH).

Power = Pin 5

Button A Signal = Pin 6 (with Select LOW).

Button B Signal = Pin 6 (with Select HIGH).

Select = Pin 7

Ground = Pin 8

Button C Signal = Pin 9 (with Select HIGH).

Start button Signal = Pin 9 (with Select LOW).

The controllers we used were a combination of team members' controllers and some that happened to be around the lab. At first we tried to connect the controllers to the ps/2 ports using a db9 to ps/2 converter. This only works for serial db9 connectors though, so this did not work for our project. Instead we used a specialized db9 connector board plug in. Two of these were soldered into a piece of perf board. These were then attached to wires to properly transmit the signals sent between the controller and the board. The controller connections were tested using a multimeter and test voltages put onto the wires. The voltages used to power the controller can be 3V, which allows the controller to interact directly with the pins on the Vertex board. The wires can then be attached to the I/O expansion pins on the board using a ribbon cable. A picture of the connection can be seen below.

Unfortunately when we came down towards the end of the project we realized that only 1 of the controllers fit on the board (We had 48 pins to work with, 32 single ended

and 16 double ended. The cartridge reader needed 40 of them. This left 8 pins, with a controller needing 7, after power and ground were taken out). So we could only attach one of the original Sega Genesis Controllers. This left game play a bit lacking, because some of the best games are played as a combo or against an opponent (See *Streets of Rage*, *Gunstar Heroes*, and *Mortal Kombat*). To fix this mess, we initially just made the buttons on the board the second controller. This was a poor fix, as it left the second player at a significant disadvantage. To fix this, we finally decided to attach a keyboard as a second controller.

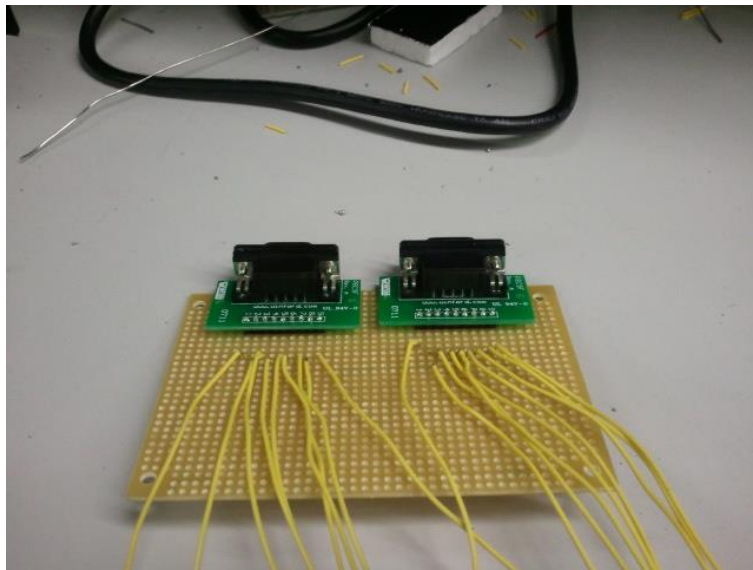


Figure 7. Sega Genesis controller breakout board.

***Keyboard / Interface:***

There are 2 ps/2 connectors on the Virtex 5 board; one for the mouse and one for the keyboard. So the keyboard is initialized when you attach it to the board. The interface to interact with the keyboard is very simple and can be seen in the picture below:

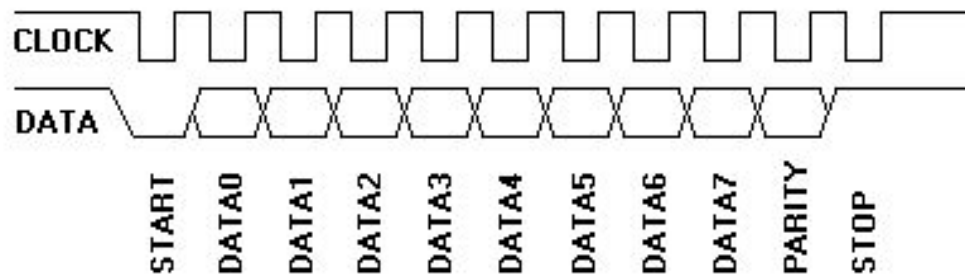


Figure 8. Timing diagram for keyboard input.

The basic idea is that the board should wait for the keyboard to initialize interactions. The keyboard communicates on two single bit lines, so the data will be sent serially. Both the clock and the data line are held high when no data is being sent. When you press a key on the keyboard, both the clock and the data line will go low. The clock will proceed to go low 10 more times. The first 8 times will be the keyboard sending the data bits. The next bit will be the odd parity bit. The final bit will be the stop bit, and it will always be high. The keyboard then will wait a certain period of time before it sends its next byte, so that the board has a chance to send a NAK (It did not get the data correctly).

The actual data that the keyboard is sending is very simplistic. For the most part, with a few exceptions, a key is represented by a single byte value (This is called the make code). When a key is pressed, that byte value is sent. The keyboard will then take one of two paths: If the key continues to be held down, the make code is sent again and again, but if the key is let go the break code is sent. The break code means that the key you pressed has been let go. The break key is normally just the make key proceeded by the byte 0xF0.

Based on this information, the keyboard works as a very good second controller. We mapped the keys Z, X, C, ENT, UP, DOWN, LEFT, RIGHT on the keyboard to the A, B, C, START, UP, DOWN, LEFT, and RIGHT on the Genesis controller.

### ***Cartridge / Interface:***

The cartridge is a 64 pin ROM with at most 4 Mbytes of memory. The 64 pins are divided into a front 32 and a back 32 (b pins and a pins). The pins correspond to the



following:

Table 1. Sega Genesis cartridge pinout.

Pin	In/out	Signal	Pin	In/out	Signal
A1	-	gnd	B1	-	?
A2	-	+5v	B2	i	!H_RESET
A3	o	a8	B3	-	?
A4	o	a11	B4	o	a9
A5	o	a7	B5	o	a10
A6	o	a12	B6	o	a18
A7	o	a6	B7	o	a19
A8	o	a13	B8	o	a20
A9	o	a5	B9	o	a21
A10	o	a14	B10	o	a22
A11	o	a4	B11	o	a23
A12	o	a15	B12	o	VIDEO
A13	o	a3	B13	o	VSYNC
A14	o	a16	B14	o	HSYNC
A15	o	a2	B15	o	HS_CLK
A16	o	a17	B16	o	!C_OE
A17	o	a1	B17	o	!C_CE
A18	-	gnd	B18	o	!AS
A19	io	d7	B19	o	CLK
A20	io	d0	B20	i	!DTACK
A21	io	d8	B21	o	?
A22	io	d6	B22	io	d15
A23	io	d1	B23	io	d14
A24	io	d9	B24	io	d13
A25	io	d5	B25	io	d12
A26	io	d2	B26	o	!LO_MEM
A27	io	d10	B27	o	!RESET

A28	io	d4	B28	o	!LDSW
A29	io	d3	B29	o	!UDSW
A30	io	d11	B30	i	!S_RESET
A31	-	+5	B31	o	?
A32	-	gnd	B32	i	!CART_IN

We were not able to find a Sega Genesis cartridge reader on the web and were unable to desolder the one found on the board. So instead we ordered a 33x2 pin cartridge reader that we use to read the cartridges. We then took the cartridge reader and soldered it into a piece of perfboard. Then, using the same process used in the controllers, we attached wires to each of the pins. We then spent a very laborious couple of hours testing the pinouts and making sure that all of the connections were good. This was, once again, done using a multimeter and some test voltages. From here we were able to connect the wires to a bigger bread board that allows us to position all of the address and data lines right next to each other in ascending order. This allows for easy connection between the address and the singled ended IO expansion pins, and between the data and the double-ended IO expansion pins. Some of the specialty signals seen above also had to be configured to power and ground in order to read the data properly. Finally, as with the controllers, the cartridge reader can be powered by 3V, so it can be attached to the IO pins safely. A picture of the cartridge reader is shown below.



Figure 9. Sega Genesis cartridge breakout board.

## TESTING AND VERIFICATION

### *FPGA Board*

We used several tools in order to debug each stage of the system. These included ChipScope, Xilinx iSim, and printf debugging for C programs. The details of our debugging issues are described above in the appropriate sections. In general, the approach we took was to first understand the component fully by reading the available documentation on it, reviewing any source code that had already been written, doing hand calculations if necessary, simulating the system or running it with ChipScope, and comparing our expected results to what we obtained from simulations or ChipScope. If needed, we would isolate a specific part of the system and generate a test bench to test it under specific conditions. The *Debugging* textbook for this class was useful for getting us in the proper mindset for debugging.

### *Hardware:*

Our first tip for testing hardware is to find suitable software parallel to test against. For example, in order to test the cartridge reader, we had to read its output and compare it to the output gotten from ROM's floating around the internet. After a lot of trouble trying to verify that the ROM themselves had the correct info, this is a very useful source to basically emulate what you are doing in hardware so that you have a "correct" source to compare against. This "correct" source is paramount when trying to test hardware, because there are so many things that can go wrong that you cannot just say that it seems like it should be working and then move on.

The main problem we found with debugging hardware was that there are a lot of little things that can go wrong that you would not think to check. If you are using solder, there is a possibility that a completely working and connected solder could break halfway through the process if it is handled or set down wrong. On top of this, if you are dealing with a large amount of wires, it is paramount to make sure that all the wires are in the correct place when you are starting to debug. This should be the first source of error that you check, because it is so easy for these wires to get out of place. To test both of these issues it is critical to have the typical Electrical Engineering resources (i.e. a multimeter,

a current source, wires, and probes). These tools and knowing how to use them are paramount to being able to debug the critical hardware errors you will face.

Finally, the major bug you are looking at is making sure that the firmware, the hardware software interaction, is set up correctly. It is just making sure that the wires you think are connecting to a certain area on the board are really connect there. If all else fails in debugging, this is where you should turn.

## **WORDS OF WISDOM / CONCLUSIONS**

First, as a nod to documentation problems, documentation is paramount to being able to design and finish your project. But just because you find a document that says a piece of hardware works a certain way, it does not mean that it actually works that way. Make sure that you verify all sources you are using to save conflicted ideas of how a certain part of your project works.

If you are thinking about using actual hardware, don't! But in all seriousness, before you take the steps to use hardware make sure you are aware of the extra stress it can cause you as well as its unpredictability. If you are aware upfront how much of a bear it might be to implement, it will better help you schedule your time and allow for extra implementation time.

The labs that are done in class are very useful. They help you learn the basics of sound, debugging with ChipScope, and how to interact with the board. Do not take these labs lightly or think that they do not matter. If you spend time initially setting up and learning these basics, they will help push what you are able to accomplish and make you more confident in attempting more challenging projects (i.e. full sound implementation seen in our project).

An entire semester to work on your project seems like an eternity the first week, but you'll be surprised at how quickly it ends. Consider the amount of time you will need to get comfortable with the tools, understand the system you are trying to build, implement it, debug it, and put everything together. Things tend to take longer than you think they will, and other classes and real life sometimes get in the way at the worst possible times. Keep a manageable schedule, stick to it as best as possible, and have a backup plan if it looks like things are falling behind near the end of the semester. But above all, pick a project that you are passionate about and want to see through to the end! More than anything, this will determine your success, enjoyment, and how much you learn from this class.

## INDIVIDUAL PAGES

*Alex Etling:*

For the Sega Genesis Project I spent the majority of my time on the externals of the system. This included the cartridge reader, the controllers, and, during the last week, attaching and writing the controller for a keyboard. I also did some work with the Programmable Sound Generator, but ultimately it was Kun who implemented it.

My time broke down into a few different phases of the work. I spent the first two weeks doing general research on the project, figuring out which role I would take and trying to figure out what parts I would need to implement the hardware. This got stretched out to two weeks as I constantly kept thinking of more parts that I would need to implement the project. Around the middle of the fourth week I had all of the parts I needed and could finally start implementing the parts that were required.

I started with the cartridge reader initially. It took a long amount of time to learn how to solder what I needed to (I had only soldered once in my life before this project), plan out how I wanted to solder the pieces together and then actually do the soldering. This process took a little over a week to solder both the cartridge interface, but also the controller interface. I then had to test all of my solders with a mutlimeter and proceed to fix anything that might have gone wrong. Finally, I could start to wire up the two interfaces on a large breadboard. This, along with some ribbon cable, allowed me to connect the 40 pins I needed to for the cartridge reader to the board. I was then, over the next 3.5 weeks, able to write a program that allowed me to interact and debug the cartridge reader. There were three main errors I had to confront in this area. The first was the difference between internet ROMs and the cartridge memory itself. The second was dealing with small wiring errors when trying to work and move over 150 wires. The third was small soldering errors that cropped up throughout the process. I was originally supposed to finish all of my hardware work by the end of the 8<sup>th</sup> week. This deadline ended up getting pushed back for to the week before Thanksgiving, when I finally finished all of the hardware.

I spent the next couple of weeks doing some in-depth research into the PSG, and gathering enough documentation that I could start it the week after Thanksgiving. When I finally got back to school and started working on it though, I was charged with the task

integrating all of my hardware into the final project and adding a keyboard interface the last week and half. This took up a good amount of time the last week, and that is why Kun ended up implementing the PSG in the end. At the end of the project all of my externals were attached and working. I was really proud of the fact that I had put so much work into building these externals from scratch and they were functioning just as expected in the end. I can look back and really appreciate all of the hard work I put in because of the final project we were able to present on demo day.

As far as time spent on the class, I probably averaged about 10 hours a week outside of classwork working on our project. This bumped up significantly the last two weeks, where I probably spent somewhere between 40-50 hours in the last two weeks working to finish my part. I think that the class was a lot of work, but that it really paid off in the end, when all of our disparate parts came together.

*Ben Joyce*

For this project, I mainly worked on getting the VDP to work and video output. After we finally got the video output to look mostly correct, I worked on fixing some additional display bugs. Here is what I did and how much time I spent:

Getting video output to display on monitor – 25 hrs

Changing video output from current VDP to display from our board – 6 hrs

Debugging VDP so video output was readable – 50 hrs

Fixing minor display bugs – 40 hrs

This project is probably the biggest project I've done so far, so it was a little intimidating at first and I think I got off to a slow start. I was expecting to be able to just use the open source VDP that we obtained and plug it into our design and then go work on other parts of the design, like sound. When it didn't work the first time, it was daunting to have thousands of lines of VHDL code to debug (especially because I was more comfortable with verilog). I had to understand the whole design to be able to see what was wrong instead of using it like a black box. This meant looking through tons of unofficial documents written by Sega enthusiasts who often contradicted each other and by the end I felt that I probably should have just written a new VDP myself. Once the major bugs were fixed and the games looked correct, fixing minor video bugs, like adding highlights and shadows or fixing vertical scrolling issues, was more approachable and it was easier to see the instant gratification in my work, which was a nice break from the earlier parts of the project.

Overall, I think this project was a fun experience and I definitely learned a lot about designing a full system. While the work sometimes seemed tedious after spending 8 hours buried in code, I definitely gained a lot of new skills and knowledge. At first the project seemed too big to handle, but once I got into it, and once I learned that debugging consists of more than waiting fifteen minutes for my code to synthesize and staring at chipscope looking for what went wrong, it was rewarding to see the progress I could make after an afternoon in the lab and kind of relaxing.



As for the parts of the class that did not have to do directly with our project, I thought they were helpful with our success. I'm not usually one to enjoy reading books for class, but the assigned texts were interesting and definitely helpful with our project. While sometimes the project got overwhelming, I think it was an overall rewarding experience and I learned a lot that I couldn't have learned in a different class.

*Kun Li*

For this project, I was primarily responsible for porting the memory modules from the Altera-based fpgagen project to our Xilinx board and for developing the sound chips and interfacing them to the rest of the system. I also helped debug issues we were having with clock stability and video output bugs.

During the beginning of the semester, I spent roughly 10 hours a week researching the Sega Genesis, tracing through the fpgagen code, and implementing changes to make it compatible with the Xilinx board. This bumped up to about 16 hours a week towards mid-semester while we debugged critical sections of the DVI controller and a major stability bug.

For the last five weeks of the semester, I spent over 40 hours a week researching the YM2612 FM sound chip, working out its implementation, simulating, debugging, and interfacing it to the rest of the system. After gaining experience with this sound chip, it was fairly straightforward to implement the PSG, which I did in 3 days during the final week. One thing I am very pleased with is the quality of the sound emulation. I tried to match the original hardware implementation as closely as possible based on the available literature. I also double checked our output closely against the original for several test games, especially Sonic the Hedgehog and Streets of Rage, which allowed me to detect several discrepancies and correct them. Some of this may have gone unnoticed, since it is more difficult for the casual user to notice bugs in the sound than in the video. However, it was very personally rewarding to create this chip accurately from scratch, as it was something that had never been done before in HDL.

Ultimately, I believe that our team had a very successful project. We had a lot of fun playing our system and seeing other people play it as well. We also added a significant contribution to the open-source emulation community, to the point that people were asking for a board to be made from our system. Finally, we learned a lot about system integration, project management, and RTL design. I am very proud of our team for doing a great job cloning the Sega Genesis in all its glory, and will always remember this project fondly.