Sega Game Gear on a Chip

Design Report

Max Thrun — Samir Silbak

Fall 2012

Contents

1 Executive Summary

The goal of this project is to reimplement the original hardware of the Sega Game Gear (GG) in a Field Programmable Gate Array (FPGA). Each digital component in the original Sega Game Gear was identified and understood based on official and 3rd party descriptions and specifications of their operation. Using these references we reimplemented a majority of the original Sega Game Gear hardware in a hardware description language (Verilog) and were successfully able to run original game ROMs on an Altera DE1 FPGA development board.

2 Background

As computer systems age it is often impossible to keep physically maintaining them. For many legacy systems you cannot buy replacement parts and finding people who have the skills to repair the original hardware is difficult. By re-implementing these systems in a Field Programmable Gate Array (FPGA) you can replace old multi-chip solutions with a single chip implementation that is maintained in software. This allows you to future proof the existing design and even extend its functionality. This project is a case study on re-implementing an old game console, the Sega Game Gear, in a FPGA.

3 Design Process

Priority	Component
1	Flash Memory Interface
2	Memory Management Unit (MMU)
3	Z80 + System RAM
4	Cartridge Mapper

Table 1: CPU & Memory Implementation Priority

Priority	Component
1	Background Tiles + VRAM
2	Video Display Processor Logic
3	Sprites
4	Controller Input
5	Sound

Table 2: Video & Sound Implementation Priority

4 Requirements / Assessment Metrics

Function	Requirement Specification	Design Verified	Device Validated
VGA Output	Design must output video at 640x480 to a VGA monitor	Yes	Yes
Sega Mapper	Design must implement the Sega Memory Mapper	Yes	Yes
Z80 CPU	Design must implement a functional equivalent to the Z80 CPU	Yes	Yes
Video Display Processor	Design must implement the TMS9918	Partly	Partly
Game ROM stored in flash	Design must be able to load game ROMs from flash	Yes	Yes
Controller Input	Design must implement a single controller	No	No
Game Gear system functionality	Design must implement the same functionality as the original Game Gear	Partly	Partly

5 Use of Standards

5.1 Verilog 2001

The core of this project is built using the Verilog Hardware Description Language defined by the IEEE standard 1364-2001 [10]. This standard describes the specifics of the Verilog language in regards to its lexical structure, data types, expressions, hierarchical structures, and its many other language features. Fortunately for us, we can depends on the Altera Quartus II compiler to ensure that all the requirements specified in the IEEE standard are met. While the full standard is rather large we only need to concern ourselves with a few sections, specifically the sections describing the syntax (2, 3, 4, 6) and overall structure of how components are modeled (5, 9, 12, 14).

5.2 GNU General Public Licensing Standard

The GNU General Public License (GPL) [11] is a standard, copyleft, license for software and other works. It guarantees end users the freedoms to use, study, share, and modify the source code of the software so long as any derivatives of the work is distributed under the same license. We felt that this license was the best choice for our project given the fact that it is primarily an academic pursuit and to be used, shared, and learned from.

6 Design Overview

6.1 System Overview

The Sega Game Gear, released in 1991, was Sega's first attempt at a handheld gaming console. It featured a Zilog Z80 clocked at 3.58MHz, 8KB of system RAM, 16KB of video ram, and a 160x144 pixel resolution screen.

Figure 1: Sega Game Gear [1]

The image below shows a picture of internal PCB of the Game Gear and it's major components

1	Zilog Z80 CPU	4	16K VRAM
2	Video Display Processor	5	Sega BIOS ROM
3	8K RAM	6	Controller IO

6.1.1 Functional Diagrams

At a high level the Game Gear can be viewed simply as a system which takes input from a game controller and produces video and audio as an output. This description is shown in Figure 2.

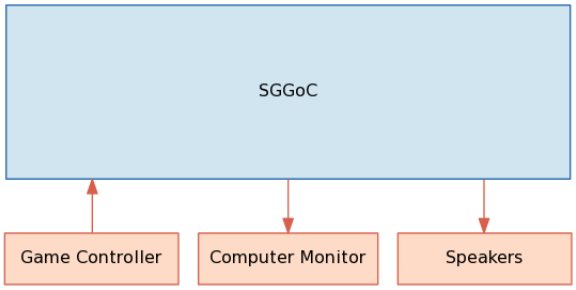


Figure 2: Black Box Diagram

Internally the functionality can be easily modularized based on the actual physical components found in the original Game Gear. Additional components will be needed such as an IO controller and a memory management unit (MMU) to account for the lack of tri-state buses in our design (explained later). Figure 3 shows the interactions between the major functional components in Sega Game Gear. The colored components are those which we implemented in our project.

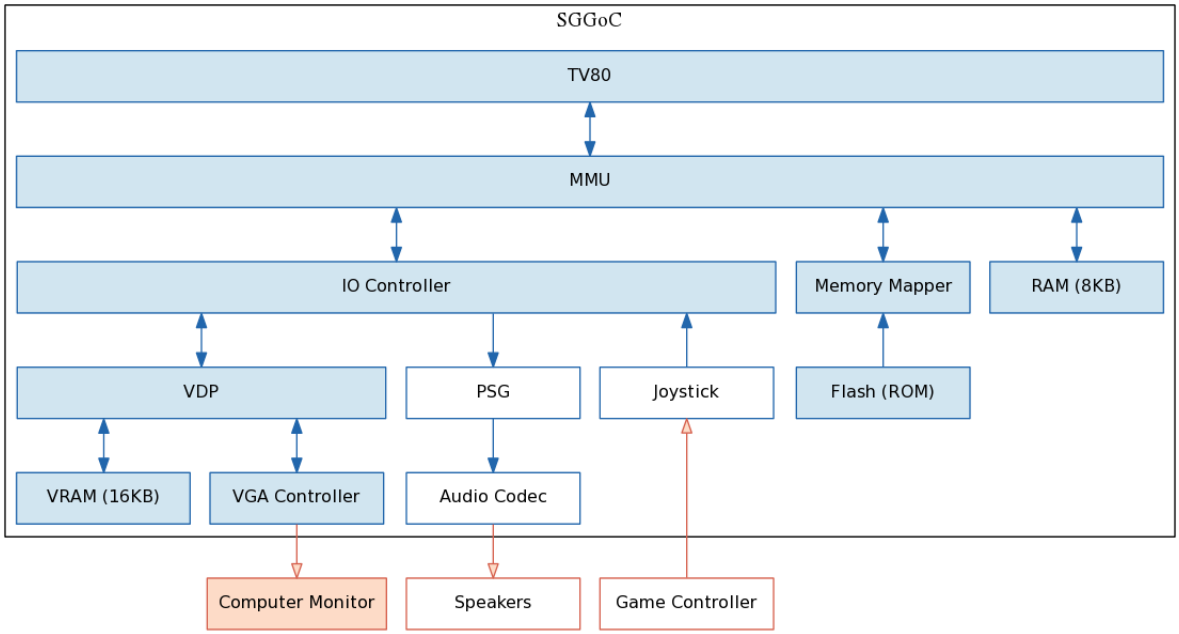


Figure 3: Internal Functional Diagram

6.2 Cartridges and Memory Mapping

The Z80 CPU only has 64KB of address space and 48KB of it is dedicated to the game cartridge. A "mapper" is used to allow the use of larger ROMs (as well as on-catridge RAM). Figure 4 shows a standard GG cartridge and Figure 5 shows the internal PCB. The top chip is the Memory Mapper and the bottom chip is the actual game ROM.

Figure 4: GG Cartridge [2]

Figure 5: Cartridge PCB [3]

6.2.1 Sega Mapper

There are several different mappers in existence but we only focused on implementing the "Sega Mapper" as it is the most popular. The Sega mapper defines 3 16KB slots in the Z80 memory map. Any 16KB bank of ROM can be mapped into any of these 3 slots. The mapping is controlled by memory addresses `0xFFFC-0xFFFF`.

Memory Address	Slot	Maps to Z80 Address
`0xFFFC`	Control Bits	-
`0xFFFD`	0	`0x0000-0x3FFF`
`0xFFFE`	1	`0x4000-0x7FFF`
`0xFFFF`	2	`0x8000-0xBFFF`

Table 3: Mapper control registers [4]

The ROM is viewed as an array of 16KB banks. The value written into the control register determines which bank is selected for that slot. Figure 6 shows an example of how the address space gets mapped to different ROM banks.

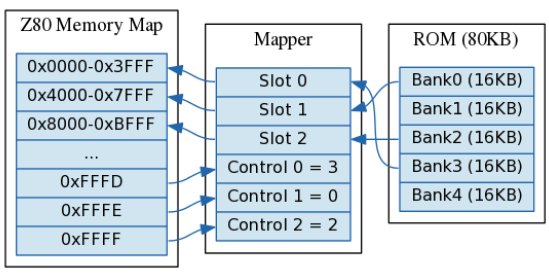


Figure 6: Mapping Diagram

6.2.2 Cartridge Alternatives

Since we are targeting a standard FPGA development board we cannot easily use the actual game cartridge without some kind of hardware adaptor. Additionally, using the actual game cartridge would go against what we are trying to accomplish with this project which is to eliminate the need for any of the original hardware. Lucky, virtually every Game Gear cartridge has been dumped and are available online. Unfortunately, the ROMs range in size all the way up to 4MB and being that most affordable FPGAs do not have that much block RAM it would be impossible for us to pack the ROM with the bitstream. A more ideal solution would be to use a SD card which can be loaded with numerous ROM files and then write a bootloader to select which one to play. The complexity of that strategy, however, was not worth the initial effort. A simpler solution which allowed us to quickly start on the rest of the project is to load a single ROM file onto the Flash chip that comes with our DE-1 development board.

6.2.3 ROM Flasher

The technical requirements of interfacing with the flash chip on our board are outside the scope of our project and as such we utilized a core provided by Altera [5] to simplify the process. Figure 7 shows the black box diagram of the flash core provided by Altera.

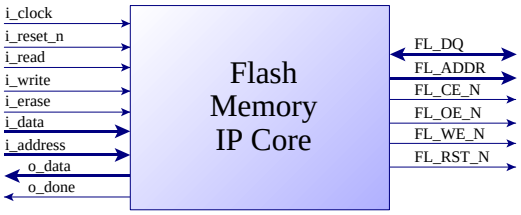


Figure 7: Altera Flash Core

A "MemSend" project, completely separate from the GG project, is used to initially load a single ROM file into the flash chip. On the computer a Python program reads a ROM file byte by byte and sends it to the FPGA over the serial port. The FPGA sends back the byte it just received as an acknowledgement. Another Python program is used to read back and verify that the flash contents match the ROM file. Figures 8 and 9 illustrate the process of writing and reading a ROM to and from the flash memory.

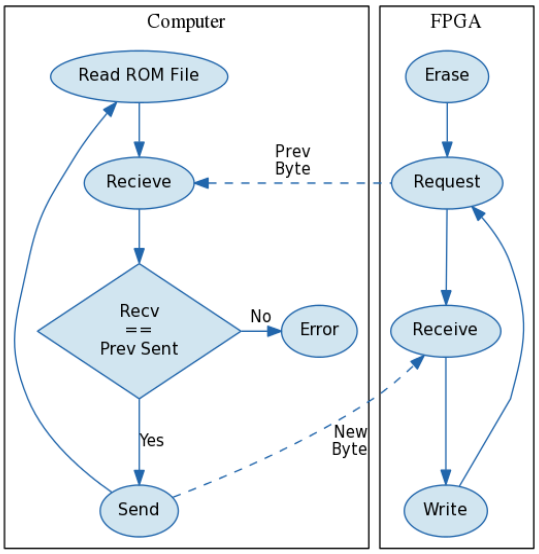


Figure 8: MemSend - Write

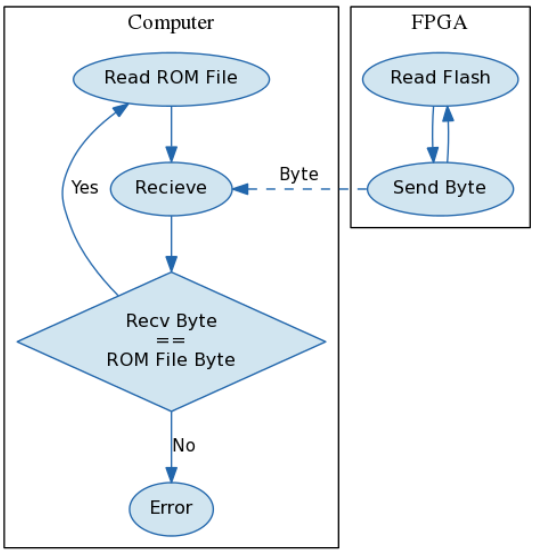


Figure 9: MemSend - Read

6.3 Zilog Z80

The Game Gear uses the classic Zilog Z80 CPU running at 3.58MHz as it's main processor. The re-implementation of the Z80 is completely outside the scope of this project and as such we utilized the popular TV80 [6] CPU which is a proven, open source, implementation of the Z80 written in Verilog. The interface of the TV80 exactly matches that of the original Z80, shown in Figure 10, with the only exception being the data bus. The original Z80, as well as the whole Game Gear memory system, relies on tri-state buses where as the TV80 has a separate bus for data in and data out.

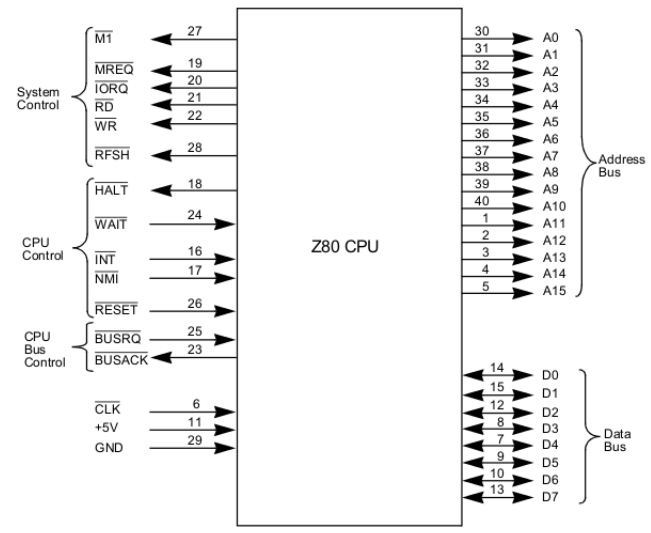


Figure 10: Zilog Z80

6.4 Memory Management Unit (MMU)

Since our design does not use tri-state buses we cannot simply connect the data lines from each component together. Instead, we need to have some kind of memory manager that can multiplex the data lines to whichever device the address is pointing to according the system memory map. The full memory map for the Game Gear is shown below

Address	Device
0x0000-0x03FF	ROM (unpaged)
0x0400-0x3FFF	ROM mapper slot 0
0x4000-0x7FFF	ROM mapper slot 1
0x8000-0xBFFF	ROM mapper slot 2 - OR - SaveRAM
0xC000-0xDFFF	System RAM
0xE000-0xFFFF	System RAM (mirror)
0xFFFC	SaveRAM mapper control
0xFFFD	Mapper slot 0 control
0xFFFE	Mapper slot 1 control
0xFFFF	Mapper slot 2 control

Table 4: Z80 Memory Map [7]

Since the cartridge mapper handles the slot mapping itself we are really only multiplexing between the cartridge, the system RAM, and the IO controller which share the memory bus. The system RAM happens to start at a nice edge being that `0xC000 = 0b1100000000000000`. To see if we are accessing RAM we only need to check to see if bits 15 and 16 are high. If so, we can simply use the 13 LSBs of the Z80 address as the index into the RAM. This method also accounts for the RAM mirror since the 13 LSBs repeat starting at 0xE000.

6.5 IO Controller

The IO Controller is responsible for determining which IO port the Z80 is currently reading or writing from during an IO operation. The port that is currently selected is decoded to one of 8 possible ports using address bits 7, 6, and 0. The following table describes which port corresponds to which IO device.

IO Port	Device
0	Memory Control
1	IO Port Control
2	VDP V Counter
3	VDP H Counter
4	VDP Data
5	VDP Control
6	Input A/B
7	Input B/Misc

Table 5: IO Port Mapping

The IO controller is also responsible for generating the VDP control and data read write signals.

6.6 Video Display Processor (VDP)

The Video Display Processor is a graphics chip which is derived from a Texas Instruments TMS9918. The VDP contains 10 control registers and interfaces with 16KB of VRAM which it reads from to continuously draw the background and sprite tiles to the screen. The Z80 can only access the VRAM through the `Data` and `Control` IO ports. Additionally, the VDP provides a vertical and horizontal pixel count which is accessed by the Z80 via the `V Counter` and `H Counter` IO ports. Finally, the VDP directly drives the interrupt line of the Z80 and asserts an interrupt when certain video events have occurred.

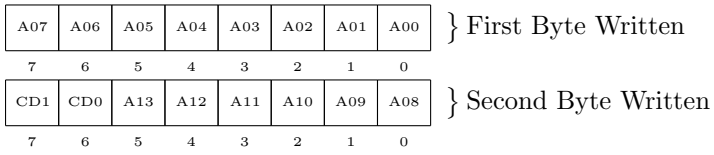
6.6.1 Control Port

The VDP is programmed by sending a two-byte sequence to the control port. The control port is used to define an offset into VRAM or CRAM for subsequent data port I/O, and also to write to the internal VDP registers.

There is a flag which is set after the first byte is sent and cleared when the second byte is written. This insures the VDP knows which byte of the control port is being received. The flag is also cleared when the control port is read or when the data port is read or written.

The address register is 14 bits in length and defines the address into VRAM for reads and writes and the address into CRAM for writes. The code register is 2 bits in length and selects four different operations.

Writes to the control port have the following format:



Where,

Bits	Definition
CDx	Code register
Axx	Address register

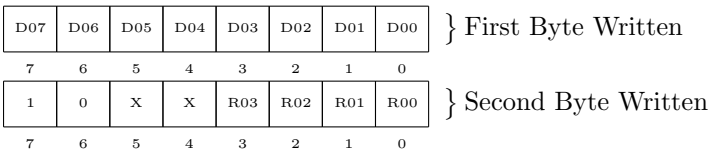
When the first byte is written it is stored in a temporary hold register. Once the second byte is written the code is checked to see if it is setting the VRAM address or doing a register write. If it is setting the VRAM then the temporary hold register is used to update the lower 8 bytes of the VRAM address and the lower 6 bites of the second byte are used to update the upper bits of the VRAM address.

The table below summarizes the different actions performed for each code.

Code	Action	Description
0	VRAM Read	A byte of VRAM is read from the location defined by the address register and is stored in the read buffer. The address register is incremented by one. Writes to the data port go to VRAM.
1	VRAM Write	Writes to the data port go to VRAM
2	Register Write	This value signifies a VDP register write, explained below. Writes to the data port go to VRAM.
3	CRAM Write	Writes to the data port go to CRAM.

When accessing CRAM, the upper bits of the address register are ignored as the CRAM is only 64 bytes. The address register will wrap when it exceeds `0x3FFF`.

If the code specified by the second byte indicates a register write the two bytes have a different format as shown below.



Where,

Bits	Definition
Rxx	Register number
Dxx	Register data
x	Ignored

The VDP selects a register using bits 3-0 of the second byte and writes the data from the first byte into that register. Since there is only 10 registers values 11 through 15 have no effect when written to.

6.6.2 Data Port

Depending on the code register, data written to the data port is sent to either VRAM or CRAM. After each write, the address register is incremented by one, and will wrap past `3FFF`.

Reads from VRAM or CRAM are buffered. Every time the data port is read (regardless of the code register) the contents of a buffer are returned. The VDP will then read a byte from VRAM at the current address, and increment the address register. In this way data for the next data port read is ready with no delay while the VDP reads VRAM.

6.6.3 Status Flags

Reading the control port returns a byte containing status flags:

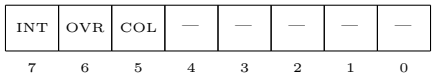


Figure 11: Status Register

The **INT** bit indicates a frame interrupt is pending and is set on the first line after the end of the active display period. It is cleared when the control port is read. (See the interrupts section for more details). The **OVR** and **COL** bits are used for sprites and are not implemented in our project. The remaining bits are not set by the VDP.

6.6.4 Color RAM (CRAM)

The Game Gear uses a 64 byte palette where each entry is composed of two bytes as shown below.

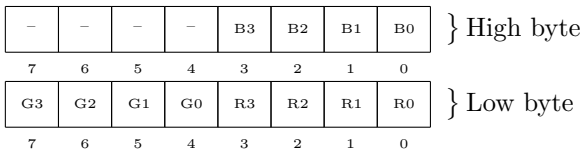


Figure 12: Palette Entry

When writing to CRAM the address register wraps at `0x0040`. Writing to an even CRAM address causes the data written to be stored in a latch, and writing to an odd CRAM address makes the VDP write the contents of the latch as well as the new data from the CPU to the current CRAM entry.

Therefore, writing single bytes to even CRAM addresses will not modify CRAM. You could write multiple times to even addresses. This allows the same latched color data to be written to multiple palette entries.

6.6.5 Display Modes

The TMS9918 has three bits which select different display modes called M1, M2, and M3. Note that in the TMS9918 manual, there are only four modes documented:

Mode 0 - Graphics I
Mode 1 - Text
Mode 2 - Graphics II
Mode 3 - Multicolor

6.6.6 Counters

6.6.7 Interrupts

6.7 VGA Controller

7 Test Plan

7.1 VDP Background Generation

In order to test the operation of the VDP background generator a separate test project was used which only consists of the background generator, VRAM, VGA timing generator, and a UART interface.

Games are run using the open-source Osmose Sega Emulator [9]. When the game reaches a screen that we want to test on the FPGA we dump the VRAM and CRAM using Osmoses built in debugger. We then use programs, written in C, that read the VRAM dump and generate images of the color palette, all 512 background tiles, and the final screen rendering. These programs serve a dual purpose allowing us to both confirm our understanding of how the background image is generated and also that our VRAM dump is valid. The images below show example output from these three programs.

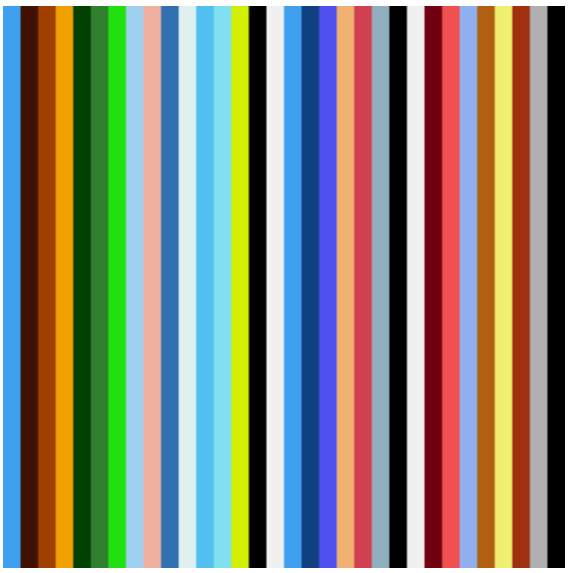


Figure 13: Color Palette

Figure 14: Complete Screen Render

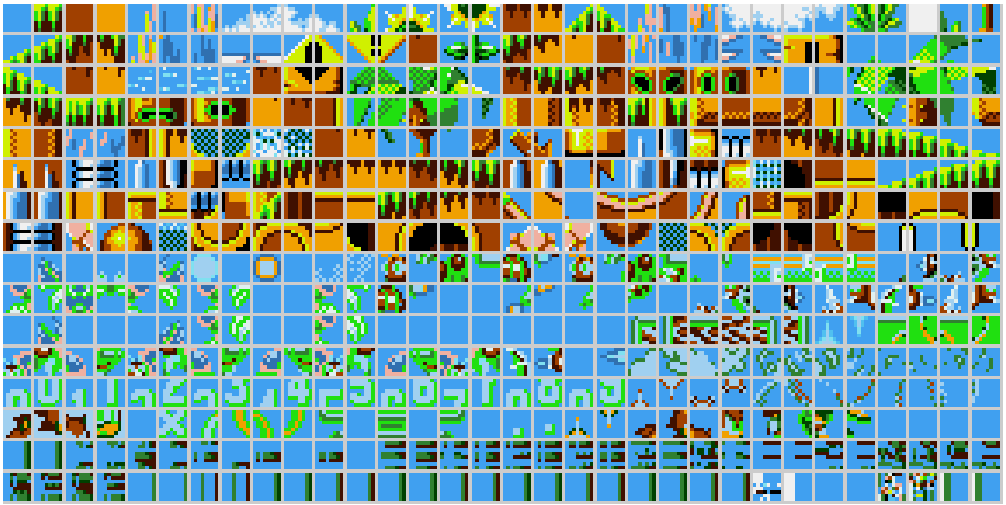


Figure 15: All 512 Tiles

After the VRAM dumps have been obtained they are loaded into the VRAM on the FPGA via the same 'MemSend' program introduced in the ROM section. The image displayed on the monitor is then verified to ensure it matches both the emulator and the output renderings of our C programs.

The testing strategy is executed as follows:

1. Obtain VRAM dump from the Osmose emulator
   (a) Launch Osmose and open ROM file

   ```
   $ ./Osmose-0-9-96-QT sonic.gg
   ```

   (b) Set a scanline breakpoint in the debug console

   ```
   Cmd: slbp 20
   ```

   (c) Continue execution by entering 'c' until the desired frame is reached.
   (d) Dump the VRAM and CRAM

   ```
   Cmd: davram 0
   ```

   (e) Dump files are located at `/tmp/osmose.(vram,cram)`
2. Verify dump integrity using C rendering programs
   (a) Run `screen`, located in `./tools/vdp_render/`, on the dump files
   (b) Ensure `screen.png` is generated
   (c) Inspect `screen.png` pixel by pixel to ensure it matches the frame displayed in Osmose
   (d) If image does not match another dump must be performed
   (e) If image still does not match then debugging needs to be performed on the `screen` rendering program
3. Transfer dump to FPGA VRAM via the 'MemSend' tool
   (a) Flash the FPGA with the Video Display Processor test project located in `./fpga/vdp_test/`
   (b) Send a ROM file to the FPGA via the `send.py` program located in `./tools/mem_send/`
   (c) Ensure the `send.py` program completes without errors
4. Verify that resulting image on screen
   (a) Inspect resulting image on monitor pixel by pixel and verify it matches both the original Osmose frame as well as the output from the C rendering programs
   (b) If the image doe not match attempt to resend the rom
   (c) If the image still does not match then debugging needs to be performed on the image rendering core.

7.2 Custom Test ROMs

In order to test the operation of the z80, and basic functionality of the VDP, custom ROMs were developed. Using the Small Device C Compiler (SDCC) [12] we are able to write programs that exercises various functionality of the system.

We found SDCC extremely easy to setup and get working. The only thing we needed to modify was the stack pointer location in the C Runtime file (crt0.s). The default stack pointer location is set to `0xFFFF` but the top most RAM address on the Game Gear is `0xDFFF`. Setting up IO is as easy as specifying a special function register at a given IO port. An example showing how to write data to the VDP data port (0xBE) is shown below.

```
__sfr __at (0xBE) vdp_data;
int main() {
    vdp_data = 0x38;     // write 0x38 to VDP data port
    while (1) {}
    return 0;
}
```

Listing 1: VDP Data Write

We ended up building a small GG library with the following functions:

```
void vdp_write_control(uint8_t value);
void vdp_write_data(uint8_t value);
void vdp_set_register(uint8_t reg, uint8_t value);
void vdp_set_vram_addr(uint16_t addr);
void vdp_set_palette(uint8_t id, uint16_t color);
void set_pattern_fill(uint16_t id, uint8_t color);
void set_tile_to_pattern(uint8_t x, uint8_t y, uint16_t pattern);
void delay(uint16_t x);
void set_debug(uint8_t x);
```

Listing 2: Game Gear Library Functions

With this library in place it is easy to to perform operations such as setting up the color palette:

```
#include <gg.h>
int main() {
    // set register 1 bit 6 to enable display
    vdp_set_register(1, (1<<6));
    // set first color palette entry to gray
    vdp_set_palette(0, 0x0CCC);
    while (1) {}
    return 0;
}
```

Listing 3: Color Palette Test ROM

A few different test ROMs were developed to test drawing tiles and scrolling. The ROMs were first run on an emulator to verify their functionality before being loaded into flash on the FPGA board. The output of the 'tiles' test ROM is shown below:

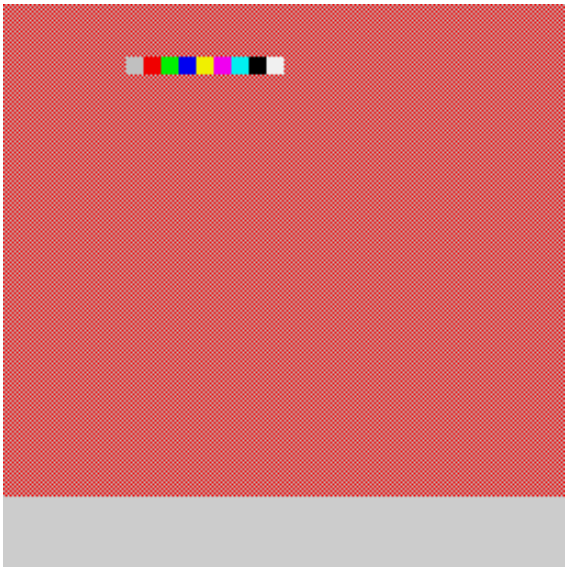


Figure 16: Tiles Screen

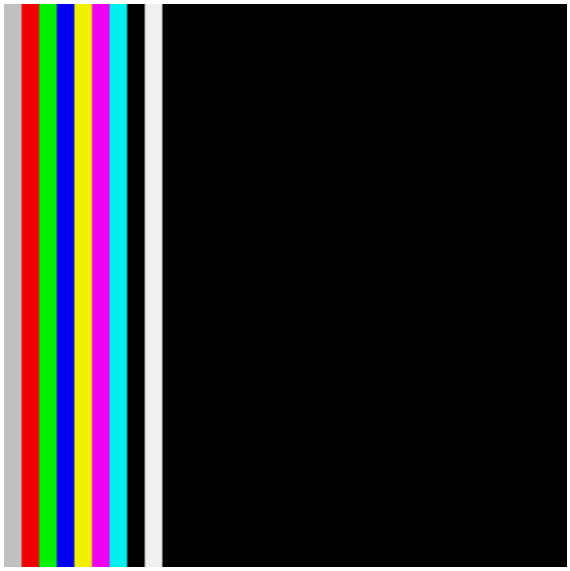


Figure 17: Tiles Palette

The general testing strategy for the test ROMs is executed as follows:

1. Identify a specific functionality that you wish to test (scrolling the background left by 5 pixels for example)
2. Write a test program that achieves the desired functionality in the simplest way possible.
3. Run the test program in the Osmose emulator and ensure the test program achieves the desired functionality.
4. Flash the test ROM onto the FPGA development board using the 'MemSend' tool.
5. Verify that the desired functionality is also achieved on the FPGA.

7.3 Simulation vs Emulator

In order to prove that our system is functionality equivalent to a known 'good' system we utilized simulations to generate dumps of all major events happening in the system. The events that we are primarily concerned with are memory and IO accesses. In order to generate logs of all the events we inserted prints into our simulation when the desired events occurred. We then inserted the exact same print into the emulator. By running the same game in both the simulation and the emulator we can compare the difference between the print logs and identify differences in functionality.

The testing strategy for this testing method is executed as follows:

1. Insert print statements on all memory and IO accesses in the emulator
2. Insert the exact same print statements on all memory and IO accesses in the simulation
3. Run the emulator and simulation on the same ROM
4. Diff the resulting logs
5. Verify that the memory and IO accesses match between the emulator and simulation

An example of a successful test result is shown below in Figure 18. Notice how both the simulation and emulator prints indicate matching functionality.

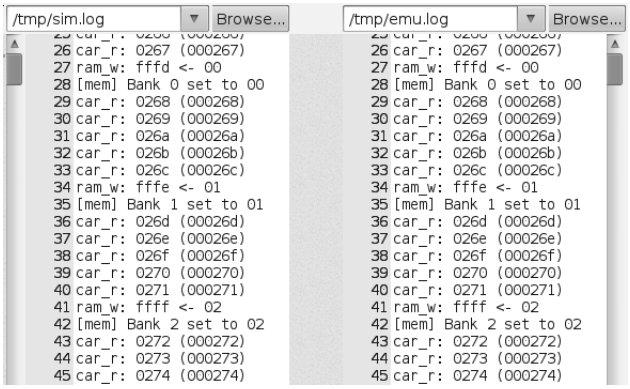


Figure 18: Successful Test Result

On the contrary Figure 19 shows an example of a bad test result. Notice how there are discrepancies between the simulation (shown on the left) and the emulator (shown on the right).

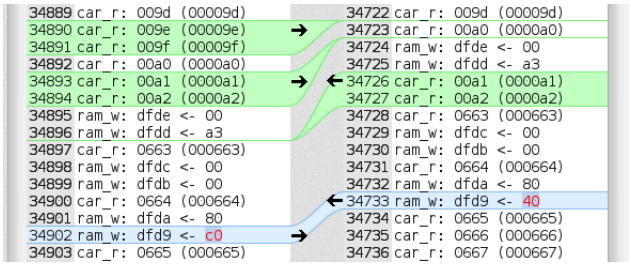


Figure 19: Successful Test Result

8 Project Limitations

Our project currently has a few limitations the most notable of which is missing core functionality. The Video Display Processor does not currently implement sprite tiles which makes most games unplayable due to the fact that the actual game characters, items, and enemies are often drawn as sprites. We also did not have time to fully implement controller support or audio support. These features are incomplete purely due to lack of time not technical problems with them. With time all components found in the original Game Gear will be implemented.

One issue that currently exists in our project is video corruption. Tiles are often flipped or the wrong tile is drawn. This corruption differs between game but seems to be static per game in that the game is corrupted the same way each time. Interestingly, some games have significantly more corruption than others. Our current guess is that there is a timing issue in the video display processor. We also believe that there might be a logic issue in the way the video display processor generates interrupts. In order to combat this issue we plan to try and create a custom test ROM which can consistently replicate the issue. We hope that we will then be able to narrow down the root cause by using similar debugging techniques discussed above such as comparing simulation results to an emulator.

An additional limitation, while not critical to the overall functionality of our project, is the fact that loading a new game ROM onto the board is cumbersome and time consuming. The method that is currently implemented was originally chosen because of it's relative simplicity in implementation and allowed us to quickly move on to the rest of the development. A better solution in the long run would be to write a simple

bootloader that loads game ROMs into RAM off an SD card. This would allow us to store multiple games on the SD card and simply choose which one we wanted to run on startup.

9 Social Responsibility

10 Conclusion

References

[1] http://mo5.com/musee-machines-gamegear.html

[2] http://www.magisterrex.com/prodimages/EccoDolphinGameGear-h450.png

[3] http://www.smspower.org/Development/SMSPagingChips

[4] http://www.smspower.org/Development/Mappers

[5] ftp://ftp.altera.com/up/pub/flash/altera_up_flash_memory.zip

[6] http://opencores.org/project,tv80

[7] http://code.google.com/p/bizhawk/source/browse/trunk/BizHawk.Emulation/Consoles/Sega/SMS/MemoryMap.Sega.cs

[8] http://www.smspower.org/uploads/Development/msvdp-20021112.txt?sid=c8bdf72dd28a0a34eeedf5f7742ca62a

[9] http://bcz.asterope.fr/osmose.htm

[10] http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00954909

[11] http://www.gnu.org/licenses/gpl.html

[12] http://sdcc.sourceforge.net/