

SECS Senior Design

Sega Game Gear on a Chip Final Design Report

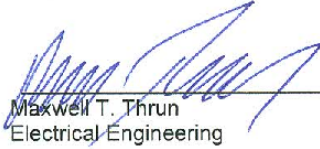
Date: 19/04/2013




University of Cincinnati, Ohio

UNIVERSITY OF CINCINNATI
SCHOOL OF ELECTRONICS & COMPUTING SYSTEMS
SENIOR DESIGN FINAL TECHNICAL DESIGN REPORT

Prepared By:



Maxwell T. Thrun
Electrical Engineering



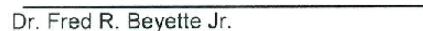
Samir H. Silbak
Electrical Engineering

Approved By:



Dr. Carla Purdy
College of Engineering & Applied Science

Approval:



Dr. Fred R. Beyette Jr.
College of Engineering & Applied Science

Contents

1	Executive Summary	5
2	Background	5
3	Design Process	5
4	Requirements / Assessment Metrics	6
5	Use of Standards	6
5.1	Verilog 2001	6
5.2	GNU General Public Licensing Standard	6
6	Design Overview	7
6.1	Sega Game Gear Overview	7
6.2	Sega Game Gear Overview	7
6.2.1	Functional Diagrams	8
6.3	Cartridges and Memory Mapping	9
6.3.1	Sega Mapper	9
6.3.2	Cartridge Alternatives	10
6.3.3	ROM Flasher	10
6.4	Zilog Z80	11
6.5	Memory Management Unit (MMU)	12
6.6	IO Controller	12
6.7	Video Display Processor (VDP)	13
6.7.1	Control Port	13
6.7.2	Data Port	14
6.7.3	Status Flags	14
6.7.4	Color RAM (CRAM)	14
6.7.5	Counters	15
6.7.6	Interrupts	15
7	Test Plan	17
7.1	VDP Background Generation	17
7.2	Custom Test ROMs	18
7.3	Simulation vs Emulator	20
8	Project Limitations	21
9	Social Responsibility	22
10	Conclusion	22
11	Code	23

1 Executive Summary

The goal of this project is to provide a case study on re-implementing a legacy computer system in a Field Programmable Gate Array (FPGA). Our project centers around re-implement the Sega Game Gear (GG) which is a hand held game console released in 1991. 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). We were able to achieve full system level simulation and proved that our system has identical execution to an emulator which we used as our known ‘good’ system to compare against. In the end we were successfully able to run original game ROMs on an Altera DE1 FPGA development board proving our project to be a success.

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

Being that our project is a re-implementation and not a completely new design in the traditional sense a majority of our design process revolved around understanding the original system, breaking it down, and understanding the components. In order to focus our efforts we attempted to split up and prioritize each of the components of the system. We determined that we could implement the CPU and it’s associated memory separate from the video and audio components. Once that was determined we prioritized the components in each subsystem based on how important they are to the overall system and their dependencies on each other.

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

Table 1: CPU & Memory Implementation Priority

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

Table 2: Video & Sound Implementation Priority

For the CPU subsystem the first step is to get a program (also called ROM throughout this document) onto the flash memory chip on our development board. Once we have a program loaded in we could then add the system RAM and the memory management unit that selects between the too. After that we were ready to implement the main CPU, the Zilog Z80. We determined the implementation of the Zilog Z80 to be outside the scope of our project and utilized an open source implementation called the TV80. At this point we were able to run custom programs and verify the functionality of the CPU subsystem.

For the Video and Sound subsystem the first thing that needed to be implemented was a VGA controller to drive a VGA computer monitor. Once that was complete the next step was to implement VRAM and the background tile render. Since the development of this subsystem was independent of the CPU subsystem we

obtained VRAM dumps from an emulator on the computer (discussed in detail in later sections) and used this dump to initialize the VRAM on the FPGA. This allowed us to work off of a known, static, VRAM instead of relying on the CPU subsystem to be correct. Basically, this technique allowed us to independently prove out the video subsystem without having to worry about the CPU subsystem being functionally correct yet. Some components such as sprites, controller input, and sound were determined to not be core functional components and given lower priority. Due to time we were unable to fully implement them.

Once the CPU and Video subsystem were independently proven the glue logic was written to tie them together. At this point we have a full system that we can test real game ROMs on. During this step we also used system level simulations to verify the overall functionality against the emulator and we were able to tweak our design based on issues found.

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
Memory Multiplexer	Memory mux must be able to properly select between the cartridge, RAM, and IO devices	Yes	Yes

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 depend 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 Sega Game Gear 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]

6.2 Sega Game Gear Overview

Our target platform for our re-implementation is the Altera DE-1 development board which contains a number of features such as a Cyclone II EP2C20F484C7, 512 KB SRAM, 4 MB Flash, VGA 4-bit DAC, and a SD card slot. There is nothing in particular about this board that makes it more suitable for our project than any other FPGA development board in the same class. It was chosen purely because we already owned them and have used them in the past.

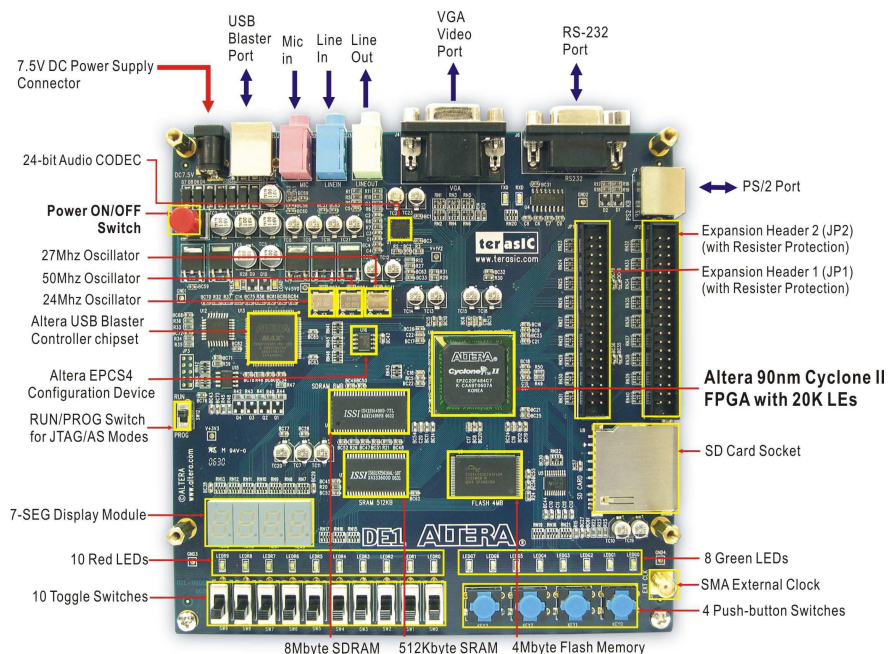


Figure 2: Altera DE1 [13]

6.2.1 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 3.

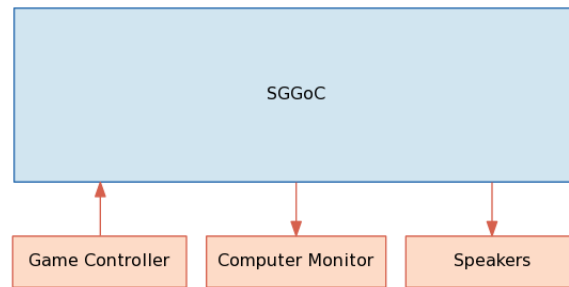


Figure 3: Black Box Diagram

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 4 shows the interactions between the major functional components in Sega Game Gear. The colored components are those which we implemented in our project.

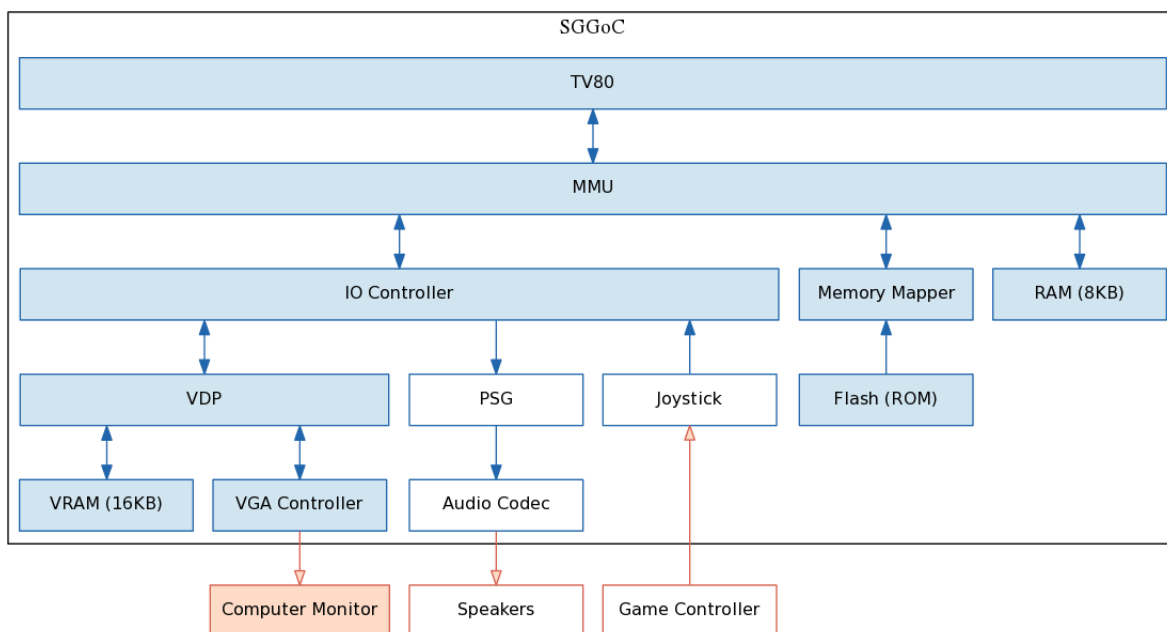


Figure 4: Internal Functional Diagram

6.3 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-cartridge RAM). Figure 5 shows a standard GG cartridge and Figure 6 shows the internal PCB. The top chip is the Memory Mapper and the bottom chip is the actual game ROM.



Figure 5: GG Cartridge [2]



Figure 6: Cartridge PCB [3]

6.3.1 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 0xFFFFC-0xFFFF.

Memory Address	Slot	Maps to Z80 Address
0xFFFFC	Control Bits	-
0xFFFFD	0	0x0000-0x3FFF
0xFFFFE	1	0x4000-0x7FFF
0xFFFFF	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 7 shows an example of how the address space gets mapped to different ROM banks.

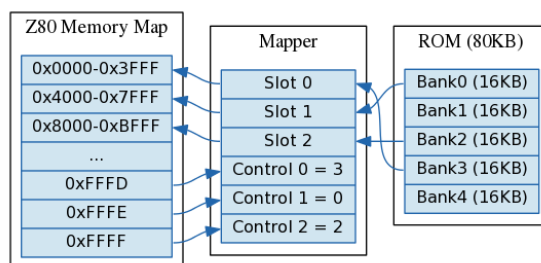


Figure 7: Mapping Diagram

6.3.2 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.

6.3.3 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 8 shows the black box diagram of the flash core provided by Altera.

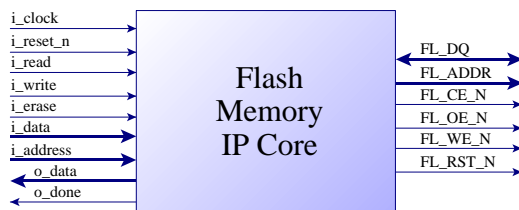


Figure 8: Altera Flash Core

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 9 and 10 illustrate the process of writing and reading a ROM to and from the flash memory.

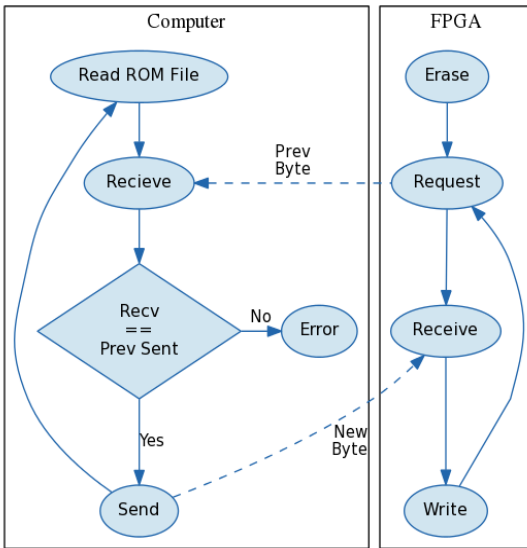


Figure 9: MemSend - Write

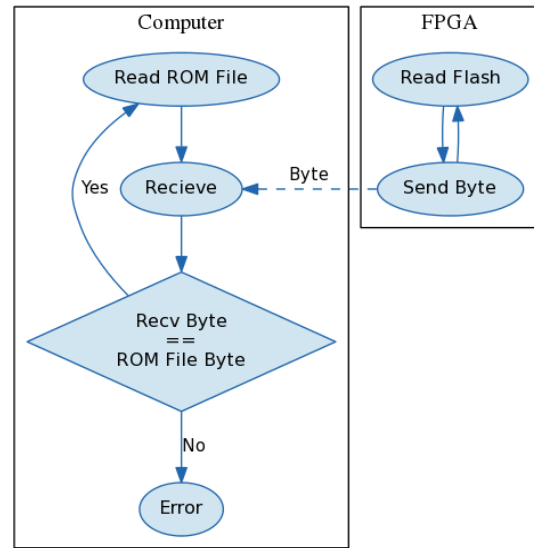


Figure 10: MemSend - Read

6.4 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 11, 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.

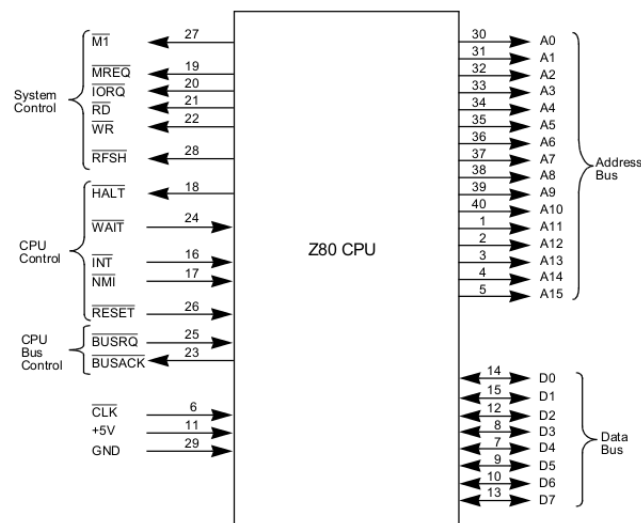


Figure 11: Zilog Z80

6.5 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 to 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 $0xE000$.

6.6 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.

6.7 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.

6.7.1 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:

A07	A06	A05	A04	A03	A02	A01	A00	} First Byte Written
7	6	5	4	3	2	1	0	
CD1	CD0	A13	A12	A11	A10	A09	A08	} Second Byte Written
7	6	5	4	3	2	1	0	

Where,

Bits	Definition
CDx	Code register
Axx	Address register

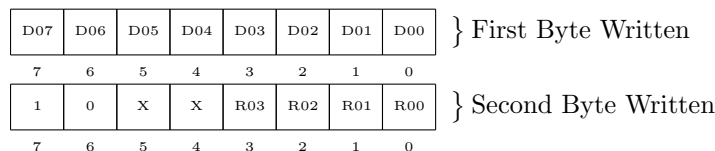
When the first byte is written it is stored in a temporary hold register. Once the second byte is written the code is checked to see if it is setting the VRAM address or doing a register write. If it is setting the VRAM then the temporary hold register is used to update the lower 8 bytes of the VRAM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

6.7.2 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.

6.7.3 Status Flags

Reading the control port returns a byte containing status flags:

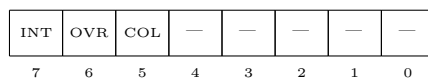


Figure 12: Status Register

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.

6.7.4 Color RAM (CRAM)

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

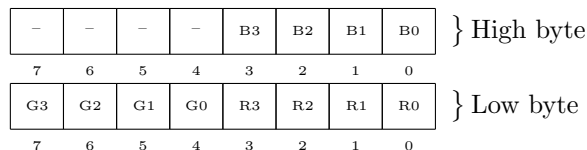


Figure 13: Palette Entry

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.

6.7.5 Counters

The VDP is responsible for both a Horizontal and Vertical counter.

The horizontal counter is incremented on the rising edge of the VGA clock. Once each scanline is complete, the vertical counter is then incremented by one, and this continues until a full frame is generated.

The horizontal and vertical counters can be read by the Z80 through the two IO ports described earlier. Our design implements the standard Game Gear NTSC display size of 256x192. The vertical counter is not completely linear. While the NTSC display size is described as 192 lines there is actually 262 lines when blanking is included. For each of the 262 lines the vertical counter increments through the following values:

V Counter Values	Line numbers
0x00 - 0xDA	0 - 218
0xD5 - 0xFF	219 - 261

Table 6: Vertical Counter Values

A description of events that occur for a given number of scanlines is shown below:

Number of Scanlines	Event
192	Active Display
24	Bottom Border
3	Bottom Blanking
3	Vertical Blanking
13	Top Blanking
27	Top Border

Table 7: Scanline Events

6.7.6 Interrupts

The VDP generates two interrupts:

1. **Frame interrupt:** occurs when vertical blanking period has started.
2. **Line interrupt:** occurs when the line interrupt counter has expired.

Frame Interrupts

The VDP support several different display modes which determine the height of the display. Depending on the display mode the VDP will generate an interrupt on a particular scanline. The table below shows which line the VDP generates an interrupt on for each of the three possible display heights.

Display Height	Scanline
192	0xC1
224	0xE1
240	0xF1

Depending on the height (fixed at 192 in our design) an event is raised at the beginning of scanline 193 (0xC1). This event sets bit 7 of the status flag register to indicate a frame interrupt request. This bit will remain set until the control port is read.

The 5th bit of the register 0 acts like an enable for the VDP's interrupt request (IRQ) line. The bits in the status register indicate an interrupt request. The VDP will only assert the IRQ line if an interrupt is requested and the 5th bit is set to a 1. The VDP will de-assert the IRQ line if either of those conditions is false.

Line Interrupts

The contents of the 10th register is loaded into a counter on every line outside of the active display period which excludes the line after the last line of the active display period. The counter is then decremented on every line within the active display period including the line after the last line of the active display period.

For example, with our design which has 192 display lines and a total of 262 scanlines per frame the counter is decremented on lines 0-192 and the counter is reloaded with the contents of register 10 on lines 193-261.

If the counter underflows from 0 to 255, then it is reloaded with the previous contents stored in register 10. Writing to the 10th register will not immediately affect/change the contents of the counter; this only occurs when the VDP is outside of the active display or when (counter is reloaded) or when the counter underflows.

The VDP sets an internal flag when the counter underflows. This flag remains set until the control port is read.

The 4th bit of register 0 acts like an enable for the VDP's IRQ line. If the internal flag is set and the 4th bit of register 0 is set, then the VDP will assert the IRQ line, and will de-assert the IRQ line if the bit is cleared.

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.

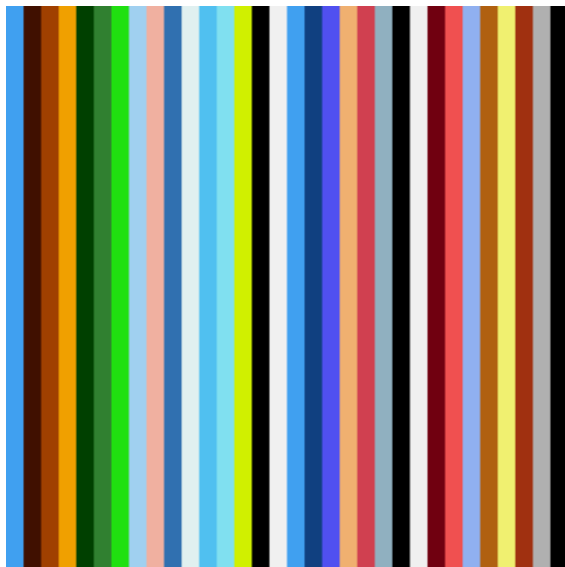


Figure 14: Color Palette

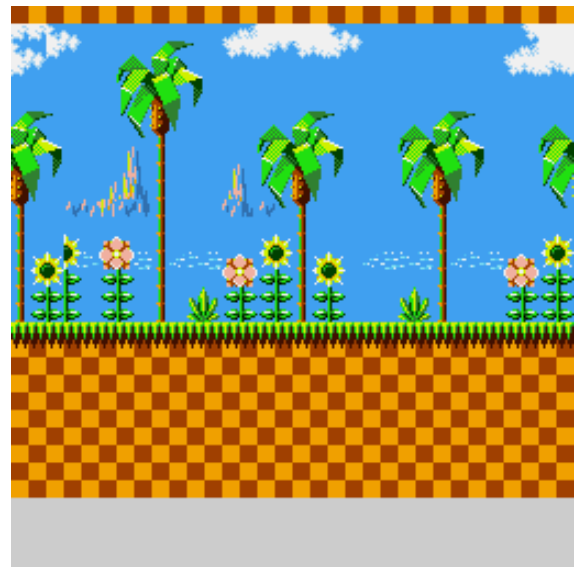


Figure 15: Complete Screen Render

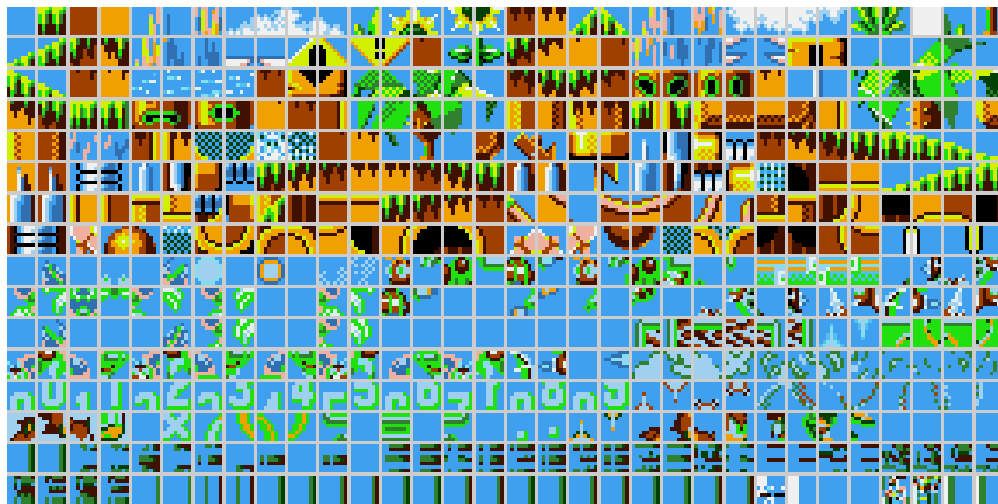


Figure 16: All 512 Tiles

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 does 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 initial stack pointer value 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.

```

1  __sfr __at (0xBE) vdp_data;
2  int main() {
3      vdp_data = 0x38;    // write 0x38 to VDP data port
4      while (1) {}
5      return 0;
6  }

```

Listing 1: VDP Data Write

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

```

1  void vdp_write_control(uint8_t value);
2  void vdp_write_data(uint8_t value);
3  void vdp_set_register(uint8_t reg, uint8_t value);
4  void vdp_set_vram_addr(uint16_t addr);
5  void vdp_set_palette(uint8_t id, uint16_t color);
6  void set_pattern_fill(uint16_t id, uint8_t color);
7  void set_tile_to_pattern(uint8_t x, uint8_t y, uint16_t pattern);
8  void delay(uint16_t x);
9  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:

```

1  #include <gg.h>
2  int main() {
3      // set register 1 bit 6 to enable display
4      vdp_set_register(1, (1<6));
5      // set first color palette entry to gray
6      vdp_set_palette(0, 0x0CCC);
7      while (1) {}
8      return 0;
9  }

```

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:

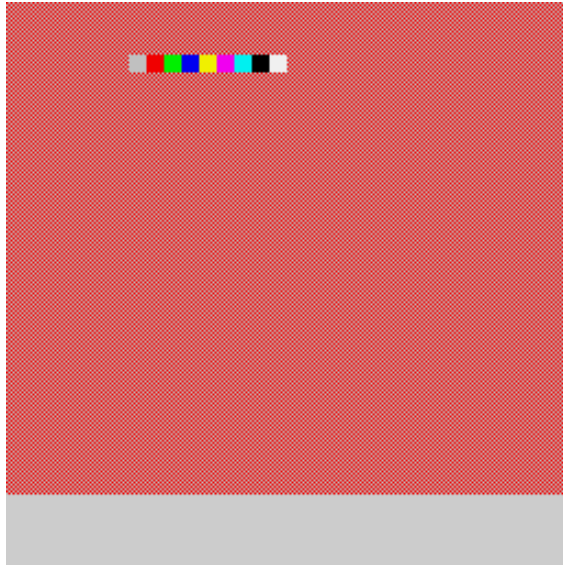


Figure 17: Tiles Screen

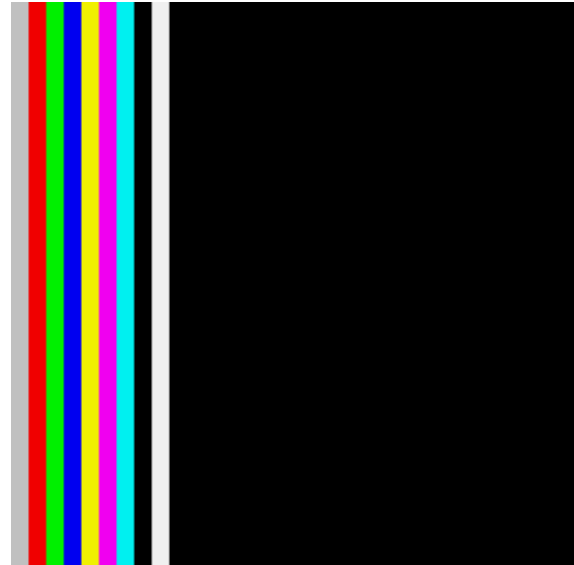


Figure 18: Tiles Palette

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 19. Notice how both the simulation and emulator prints indicate matching functionality.

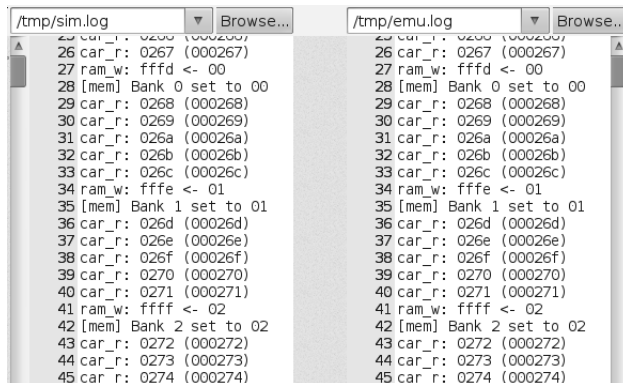


Figure 19: Successful Test Result

On the contrary Figure 20 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).

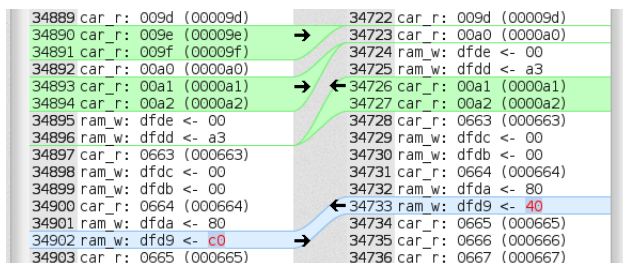


Figure 20: Successful Test Result

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.

Another issue that currently exists in our project is video corruption. Tiles are sometimes 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

Copyright and liscensing were two concerns for us throughout this project. Part of our project deals with using game ROMs that we downloaded off the internet which raises a copyright issue. While it is illegal to create a full duplicate of the original media it is generally understood that possessing a ROM dump of a game you physically own is acceptable. For our project we tried to only use game ROMs of games that we physically own. Two other components of our project, the TV80 CPU and the Osmose emulator, are open sourced under the GPL license. One of the main objective of the license is to ensure that derivatives of the project are also released as open source. This was a non-issue for us since our entire project shares the GPL license as well.

Reasonable cost and maintainability were two of the main driving factors of our project. The primary idea of our project is to show that it is beneficial to re-implement legacy systems in an FPGA. We believe that re-implementation can be done at a reasonable cost being that the physical hardware needed for an FPGA solution is cheaper than it has ever been before and the only other cost overhead is development man-hours. The cost trade off between re-implementation and monetary loss due to a legacy system failing swings towards re-implementation being more beneficial. Re-implementation in a FPGA also opens the door to better maintainability given that the system is no longer dependant on physical, deprecated, components.

10 Conclusion

In this project we showed possible methods and testing procedures for reimplementing a legacy computer system. We found that it was relatively easy to get the base system up and running which we attribute mostly to the fact that one of the core components, the Zilog Z80 CPU, already existed as a proven open source project. Trying to fully implement the Z80 would have been a huge undertaking on its own and we doubt we would have even come close to implementing the actual Game Gear system in the given time frame if we did not leverage the TV80 project. We found that translating old design concepts, such as tri-state buses, to new ones was a non-issue and our first attempt at implementing the memory and IO multiplexer was successful. Using system level simulation to verify the overall functionality of our design proved extremely beneficial in identifying irregularities between our system and the emulator that we tested against. Had we not had a proper emulator this project would have been significantly more difficult. Using the emulator as a known ‘good’ system we were able to create video RAM dumps which we used to initialize the video sub system while developing it. Had we not had this ability we would have had to develop the video subsystem blind, not knowing if any video issues were because of the CPU subsystem or the video subsystem. While there are still a few functional limitations such as no sprites or controller input we were successfully able to get commercial game ROMs running on our system proving that our re-implementation was successful.

11 Code

All code and other documentation associated with this project can be found here:
<https://github.com/bear24rw/sggoc>

References

- [1] <http://mo5.com/musee-machines-gamegear.html>
- [2] <http://www.magisterrex.com/prodimages/EccoDolphinGameGear-h450.png>
- [3] <http://www.smspover.org/Development/SMS PagingChips>
- [4] <http://www.smspover.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.smspover.org/uploads/Development/msvdp-20021112.txt?sid=c8bdf72dd28a0a34eedf5f7742ca62a>
- [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/>
- [13] <http://www.altera.com/education/univ/materials/boards/de1/unv-de1-board.html>