Sega Game Gear on a Chip

Design Report

Max Thrun — Samir Silbak

Fall 2012

Contents

1 Abstract

The objective of this project is to reimplement the hardware of the Sega Game Gear (GG) in a Field Programmable Gate Array (FPGA). To do this each chip in the system will be reimplemented in Verilog based on official and 3rd party descriptions and specifications on how they operate. This document details the design functionality and testing metrics that will be followed during the implementation and execution of our project.

2 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

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 has an output. This description is shown in Figure 2.

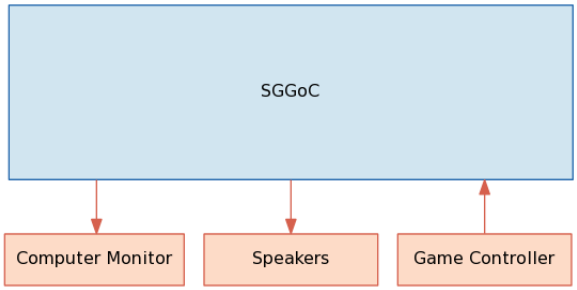

Figure 2: Black Box Diagram

Internally the functionality can be easily modularized based on the actual physical components. 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 major functional components in our design and their interaction with each other.

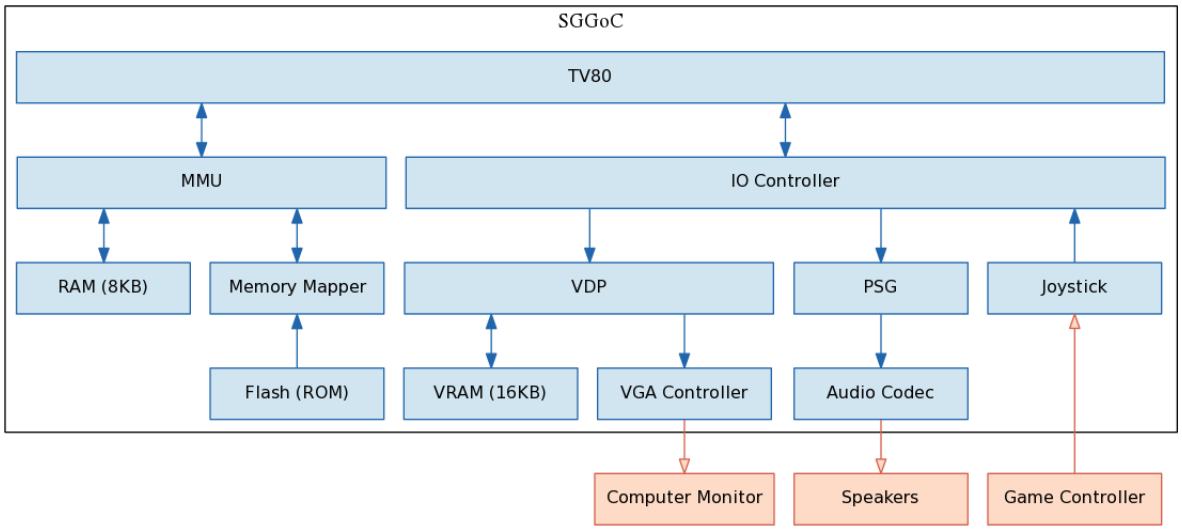

Figure 3: Internal Functional Diagram

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-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]

3.1 Sega Mapper

There are several different mappers in existence but we will only focus 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.

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

Table 1: 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.

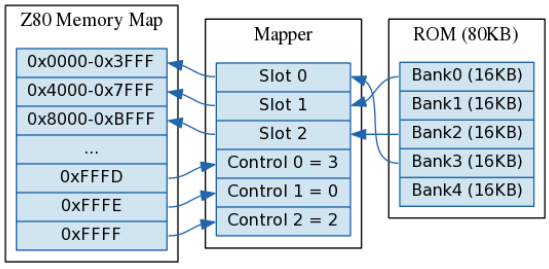

Figure 6: Mapping Diagram

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 eliminating 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, is not worth the initial effort. A simpler solution which would allow 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.

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 will be using a core provided by Altera [5] to simplify the process. Figure 7 shows the black box diagram of the flash core provided by Altera.

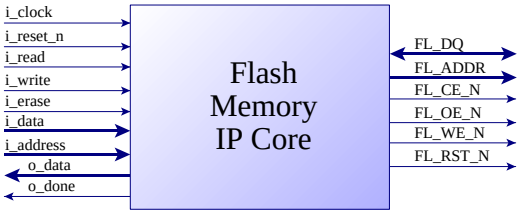

Figure 7: Altera Flash Core

A "MemSend" project, completely separate from the GG project, will be used to initially load a single ROM file into the flash chip. On the computer a Python program will read a ROM file byte by byte and send it to the FPGA over the serial port. The FPGA will send back the byte it just recieved as an acknowledgement. Another Python program will be 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.

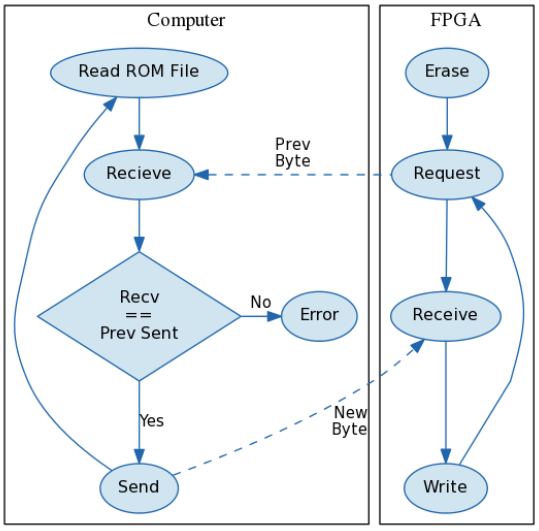

Figure 8: MemSend - Write

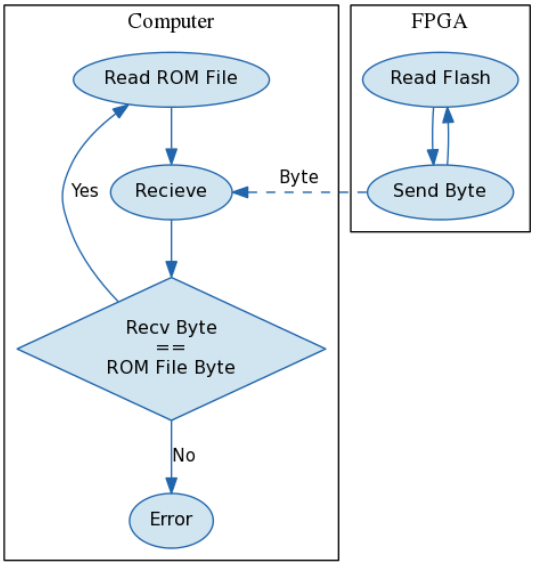

Figure 9: MemSend - Read

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 will be using 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.

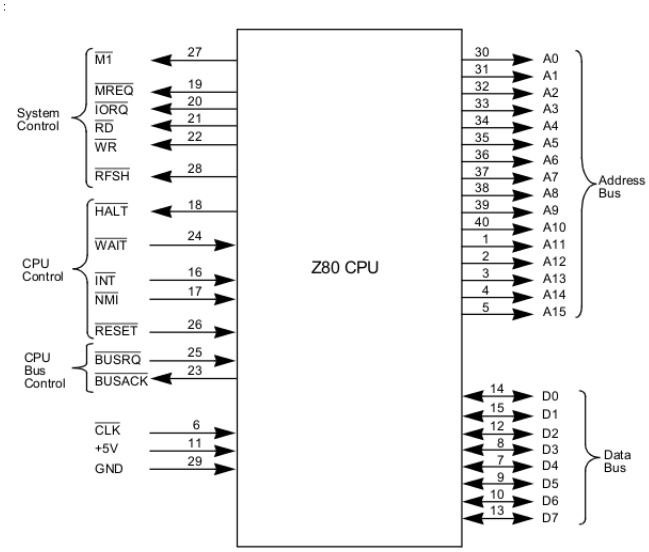

Figure 10: Zilog Z80

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 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 2: Z80 Memory Map [7]

Since the cartridge memory mapper handles the slot mapping itself we are really only multiplexing between the cartridge and the system RAM. 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 IO Controller

7 Video Display Processor (VDP)

The Video Display Processor is a graphics chip which is derived from Texas Instruments (TMS9918).

7.1 Ports

The VDP is accessed at the following Z80 I/O ports:

$7E	V counter (read) / SN76489 data (write)
$7F	H counter (read) / SN76489 data (write, mirror)
$BE	Data port (r/w)
$BF	Control port (r/w)

The address decoding for the I/O ports is done with A7, A6 and A0 of the Z80 address bus, so the VDP locations are mirrored:

- $40 - 7F = Even locations are V counter/PSG, odd locations are H counter/PSG

- $80 - BF = Even locations are data port, odd locations are control port.

7.2 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 to know which byte of the control port is being received. Note that the flag is also cleared when the control port is read, and when the data port is read or written. This is used to initialize the flag to zero after it has been modified unpredictably, after an interrupt routine has executed, for example.

The VDP has two components that are used for accessing CRAM and VRAM:

- address register
- code register

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:

- VRAM write
- VRAM read
- CRAM write
- VDP register write

The control port has the following format:

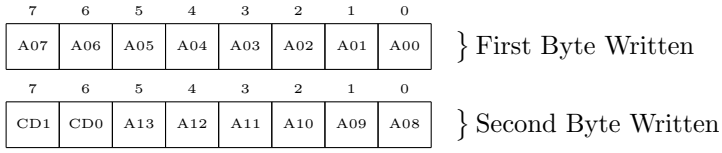

Figure 11: Caption Here

CDX	Code register
AXX	Address register

When the first byte is written, the lower 8 bits of the address register are updated. When the second byte is written, the upper 6 bits of the address register and the code register are updated, and the VDP may carry out additional processing based on the value of the code register:

Code value	Actions taken
0	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	Writes to the data port go to VRAM
2	This value signifies a VDP register write, explained below. Writes to the data port go to VRAM.
3	Writes to the data port go to CRAM.

When accessing CRAM, the upper bits of the address register are ignored as CRAM is smaller than 16k (either 32 or 64 bytes depending on the VDP). The address register will wrap when it exceeds $3FFF.

VDP register write

While the address and code register are updated like normal when a VDP register write is done, the control port sent can be viewed as having a different format to the programmer:

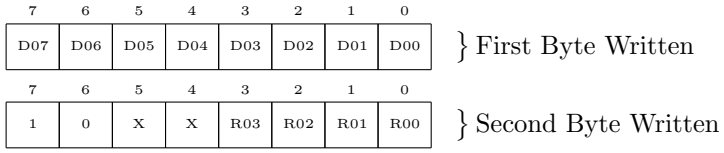

Figure 12: Caption Here

RXX	VDP register number
DXX	VDP 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 to the register in question. There are only 10 registers, values 11 through 15 have no effect when written to.

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. An additional quirk is that writing to the data port will also load the buffer with the value written.

7.3 Status Flags

Reading the control port returns a byte containing status flags:

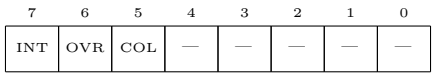

Figure 13: Caption Here

INT - Frame Interrupt Pending

This flag is set on the first line after the end of the active display period. It is cleared when the control port is read. (Please see the interrupts section for more details).

OVR - Sprite Overflow

This flag is set if there are more than eight sprites that are positioned on a single scanline. It is cleared when the control port is read. (Please see the sprites section for more information).

COL - Sprite Collision

This flag is set if an opaque pixel from any two sprites overlap. It is cleared when the control port is read. (Please see the sprites section for more information).

The remaining bits are not set by the VDP.

7.4 Color RAM (CRAM)

In Game Gear mode, CRAM has been expanded to 32 words or 64 bytes. Each word defined a single color as shown below:

-	=	Unused
R	=	Red pixels
G	=	Green pixels
B	=	Blue pixels

There are a total of 4096 (2^12) possible colors that can be used, but only 32 colors can be shown at any given time.

The address register now wraps past address $003F. 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. For example:

```
ld      hl, $C000       ; CRAM address $0000
rst     10h             ; Assume this functions sets VDP address
ld      a, $FF          ; Color data
out     ($BE), a        ; CRAM unchanged, latch = $FF
ld      hl, $C021       ; CRAM address $0021
rst     10h             ; Set the address again
ld      a, $0F          ; Color data
out     ($BE), a        ; CRAM word at $0020 is now $0FFF, and the
                        data at $0000 is unchanged.
```

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.

7.5 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

8 Programmable Sound Generator (PSG)

9 VGA Controller

10 Testing Strategies

10.1 VDP Background Generation

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

A VRAM dump will be obtained from the open-source Osmose Sega Emulator [9]. Programs, written in C and running on the computer, will 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 figures below show the three output images of these programs.

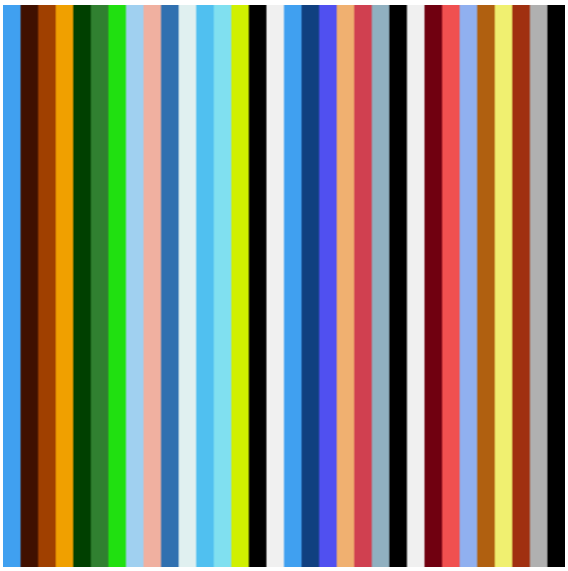

Figure 14: Color Palette

Figure 15: Complete Screen Render

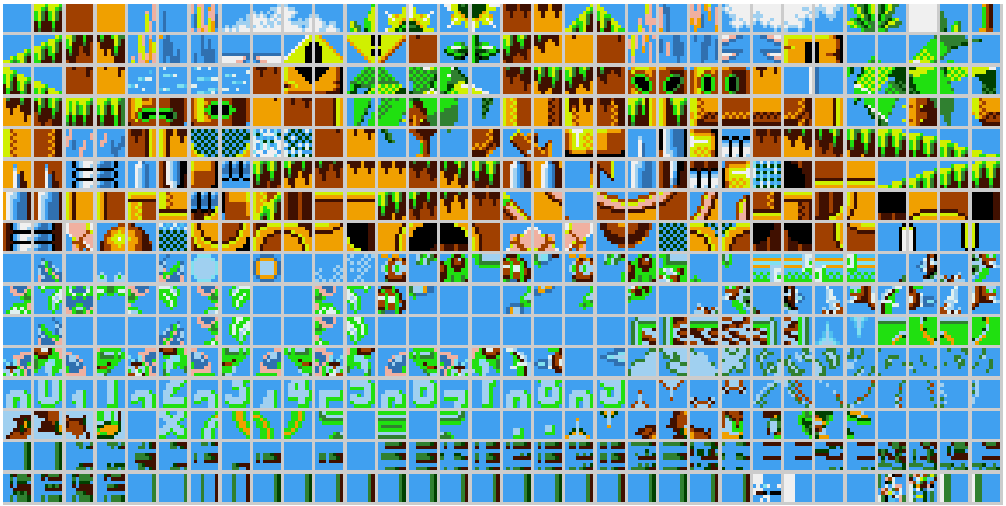

Figure 16: All 512 Tiles

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
   (b) Step the CPU in the debugger until desired frame is reached
   (c) Enter `davram 0` into the debugging console
   (d) 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' tools
   (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.

11 Standards

11.1 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).

11.2 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.

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