Game Boy Advance Architecture

A Practical Analysis

Rodrigo Copetti

Game Boy Advance Architecture

Architecture of Consoles: A Practical Analysis, Volume 7

Rodrigo Copetti

Published by Rodrigo Copetti, 2019.

GAME BOY ADVANCE ARCHITECTURE

First edition. August 18, 2019.

ISBN: 979-8201628505

Written by Rodrigo Copetti.

Also by Rodrigo Copetti

Architecture of Consoles: A Practical Analysis

NES Architecture

Game Boy Architecture

Mega Drive Architecture

SNES Architecture

Sega Saturn Architecture

PlayStation Architecture

Game Boy Advance Architecture

Nintendo 64 Architecture

Dreamcast Architecture

GameCube Architecture

Wii Architecture

PlayStation 2 Architecture

Xbox Architecture

Nintendo DS Architecture

Master System Architecture

PC Engine / TurboGrafx-16 Architecture

Virtual Boy Architecture

PSP Architecture

PlayStation 3 Architecture

Xbox 360 Architecture

Wii U Architecture

GAME BOY ADVANCE ARCHITECTURE

One chip to rule them all

Rodrigo Copetti

2019-08-18

1 ABOUT THIS EDITION

This edition originates from the article initially published on my personal website, it's been re-styled to take advantage of the capabilities of PDF/eBook documents.

While identical content-wise, interactive widgets have been simplified to work with a static environment - in other words, anything that physical pages allow us :), though these will offer a link to the original article in case the reader wants to try the 'full version'. Please keep this in mind when you see references to interactivity throughout the writings.

As always, the original manuscript of the articles is available on Github to enable readers to report mistakes or propose changes. There's also a supporting reading list available to help understand the series. The author also accepts donations to help improve the quality of current articles and upcoming ones.

2 A QUICK INTRODUCTION

The internal design of the Game Boy Advance is quite impressive for a portable console that runs on two AA batteries.

This console will carry on using Nintendo’s *signature* GPU. Additionally, it will introduce a relatively new CPU from a British company that will surge in popularity in the years to come.

3 SUPORTING IMAGERY

3.1 Model

Figure 3.1: The original Game Boy Advance. Released on 21/03/2001 in Japan, 11/06/2001 in America and 22/06/2001 in Europe. [1]

3.2 Motherboard

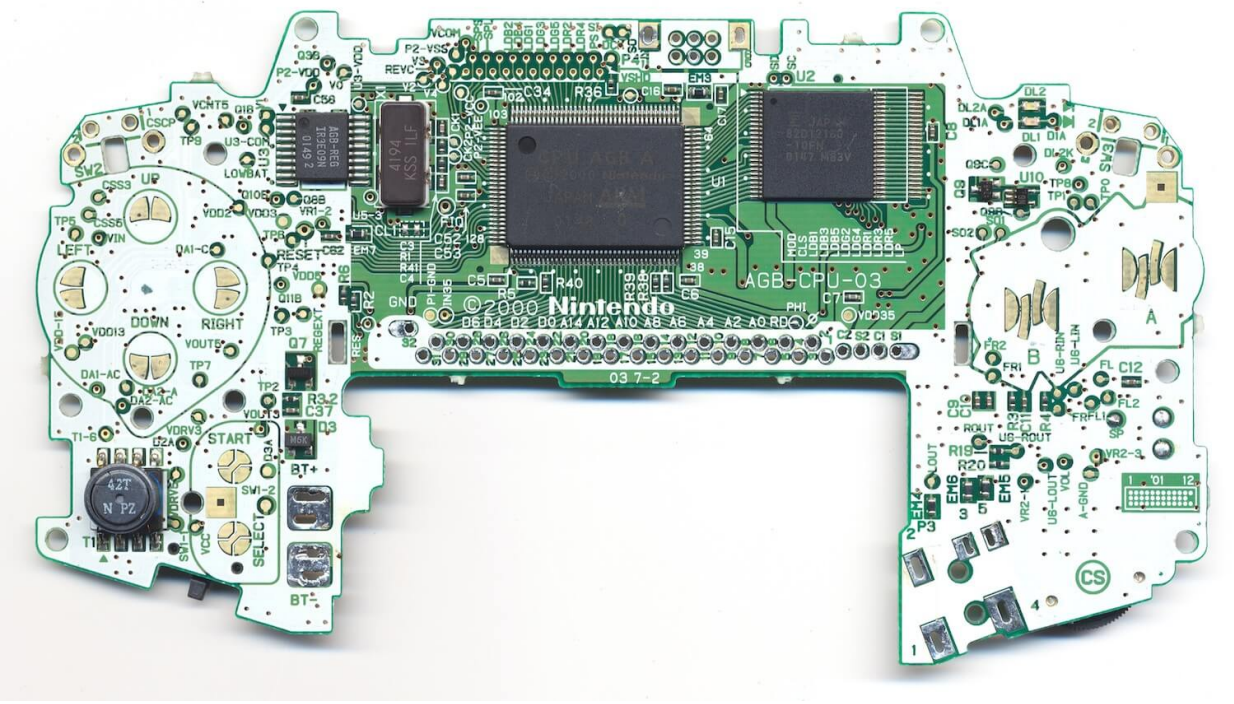

Figure 3.2: Motherboard. Showing revision ‘03’. Note that ‘AGB’ is the identifier of the Game Boy Advance model. Cartridge slot and audio amplifier are on the back. [2]

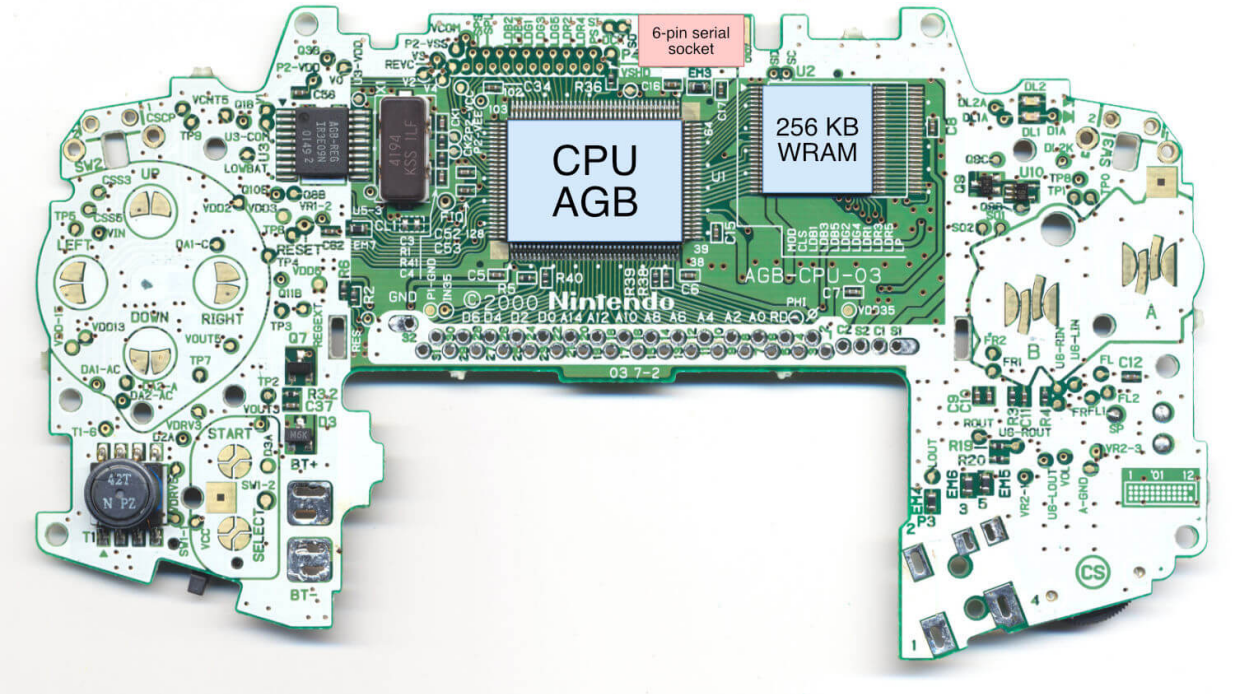

Figure 3.3: Motherboard with important parts labelled.

3.3 Diagram

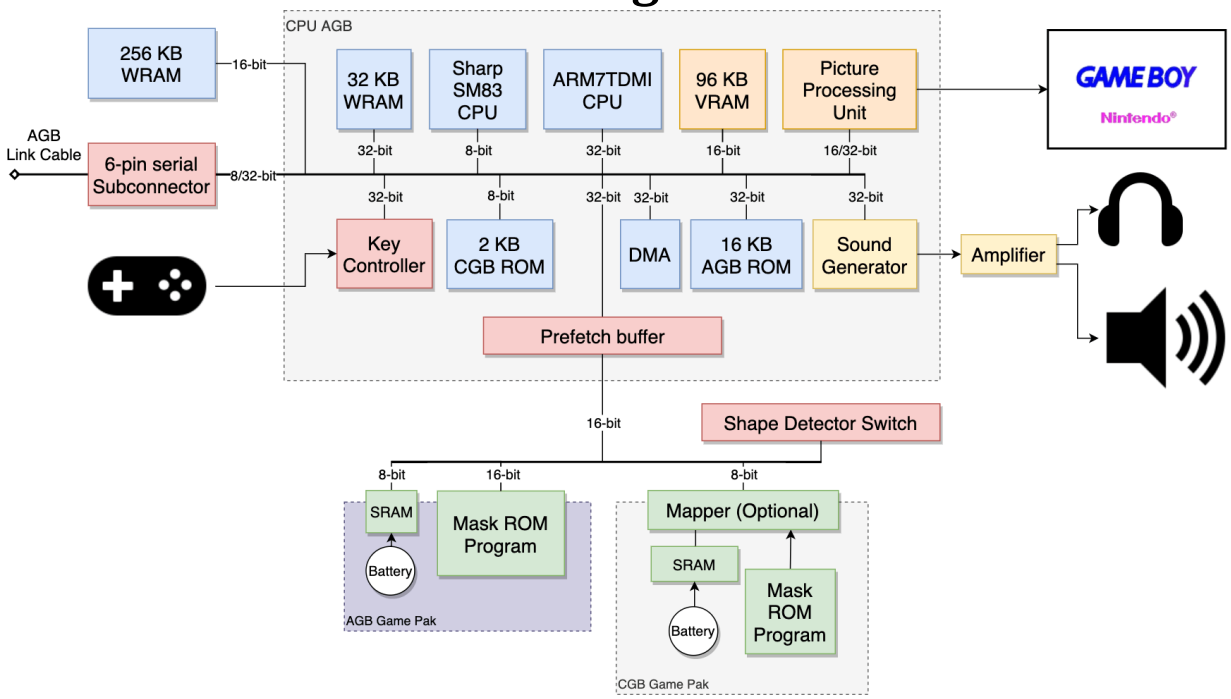

Figure 3.4: Main architecture diagram.

4 CPU

Most of the components are combined into a single package called **CPU AGB**. This package contains two completely different CPUs:

- A **Sharp SM83** running at either 8.4 or 4.2 MHz: *If it isn't the same CPU found on the Game Boy!* It's effectively used to run Game Boy (**DMG**) and Game Boy Color (**CGB**) games. Here's my previous article if you want to know more about it.
- An **ARM7TDMI** running at 16.78 MHz: This is the new processor we'll focus on, it most certainly runs Game Boy Advance games.

Note that both CPUs will **never run at the same time** or do any fancy co-processing. The **only** reason for including the *very* old Sharp is for **backwards compatibility**.

4.1 What's new?

Before ARM Holdings (currently "Arm") became incredibly popular in the smartphone world, they licensed their CPU designs to power Acorn's computers, Apple's Newton, Nokia's phones and the Panasonic 3DO. Nintendo's chosen CPU, the ARM7TDMI, is based on the earlier ARM710 design and includes [3]:

- **ARM v4** ISA: The 4th version of the 32-bit ARM instruction set.
- **Three-stage pipeline**: Execution of instructions are divided into three steps or *stages*. The CPU will fetch, decode and execute up to three instructions concurrently. This enables maximum use of the CPU's resources (which reduces idle silicon) while also increasing the number of instructions executed per unit of time.
- **32-bit ALU**: Can operate 32-bit numbers without consuming extra cycles.

Moreover, this core contains some extensions referenced in its name (*TDMI*):

- **T → Thumb**: A subset of the ARM instruction set whose instructions are encoded into 16-bit words [4].
    - Being 16-bit, Thumb instructions require half the bus width and occupy half the memory. However, since Thumb instructions offer only a functional subset of ARM you may have to write more instructions to achieve the same effect.
    - Thumb only offers conditional execution on branches, its data processing ops use a two-address format, rather than three-address, and it only has access to the bottom half of the register file.
    - In practice Thumb uses 70% of the space of ARM code. For 16-bit wide memory Thumb runs *faster* than ARM.
    - If required, ARM and Thumb instructions can be mixed in the same program (called *interworking*) so developers can choose when and where to use each mode.
- **D → Debug Extensions**: Provide JTAG debugging.
- **M → Enhanced Multiplier**: Previous ARM cores required multiple cycles to compute full 32-bit multiplications, this enhancement reduces it to just a few.
- **I → EmbeddedICE macrocell**: Enables hardware breakpoints, watchpoints and allows the system to be halted while debugging.

4.2 Memory locations

The inclusion of Thumb in particular had a strong influence on the final design of this console. Nintendo mixed 16-bit and 32-bit buses between its different modules to reduce costs while providing programmers with the necessary resources to optimise their code.

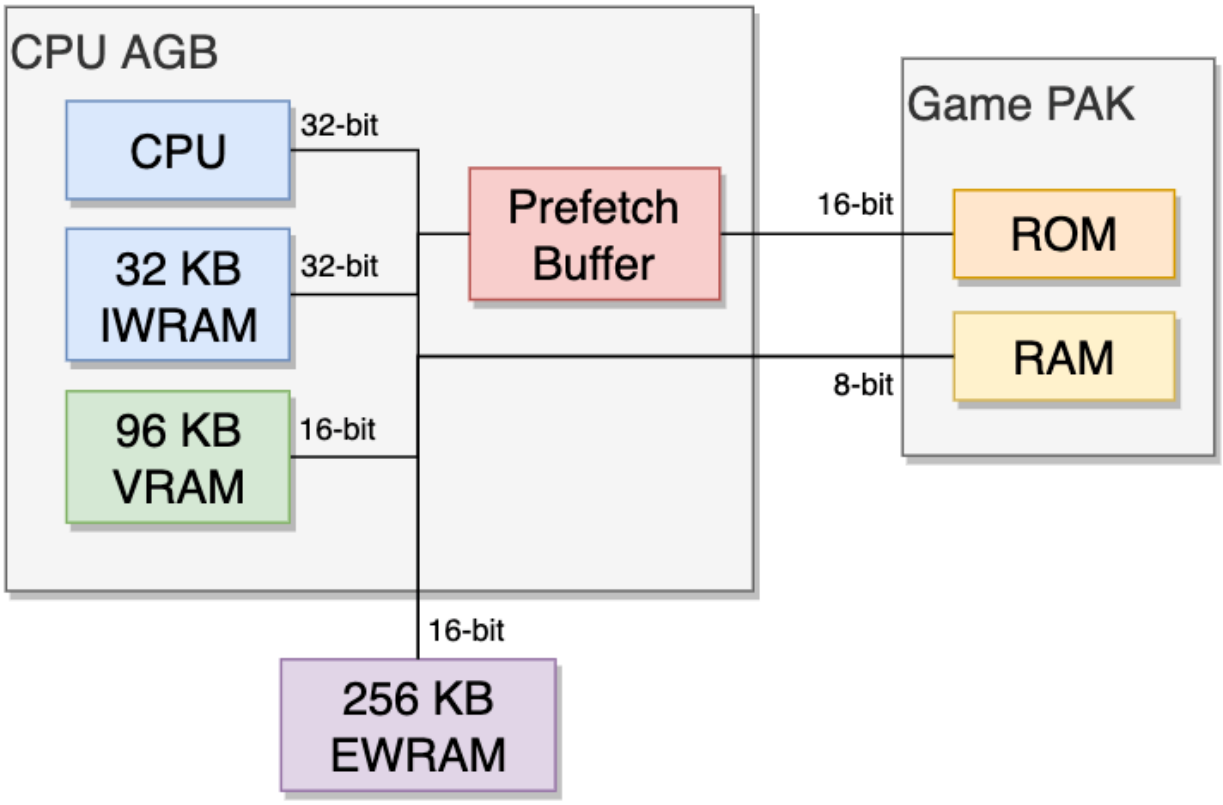

Figure 4.1: Memory architecture of this system.

Usable memory is distributed across the following locations (ordered from fastest to slowest) [5]:

- **IWRAM** (Internal WRAM) → 32-bit with 32 KB: Useful for storing ARM instructions.
- **VRAM** (Video RAM) → 16-bit with 96 KB: While this block is dedicated to graphics data (explained in the next section of this article), it's still found in the CPU's memory map, so programmers can store other data if IWRAM is not enough.
- **EWRAM** (External WRAM) → 16-bit with 256 KB: A separate chip next to CPU AGB. It's optimal for storing Thumb-only instructions and data in small chunks. On the other side, the chip can be up to six times slower to access compared to IWRAM.

- **Game PAK ROM** → 16-bit with variable size: This is the place where the cartridge ROM is accessed. While it may provide one of the slowest rates, it's also mirrored in the memory map to manage different access speeds. Additionally, Nintendo fitted a **Prefetch Buffer** that interfaces the cartridge to alleviate excessive stalling. This component independently caches continuous addresses when the CPU is not accessing the cartridge, it can hold up to eight 16-bit words.
    - In practice, however, the CPU will rarely let the Prefetch Buffer do its job. Since by default it will keep fetching instructions from the cartridge to continue execution [6] (hence why IWRAM and EWRAM are so critical).
- **Game PAK RAM** → 8-bit with variable size: This is the place where the cartridge RAM (SRAM or Flash Memory) is accessed.
    - This is strictly an 8-bit bus (the CPU will see 'garbage' in the unused bits) and for this reason, Nintendo states that it can only be operated through their libraries.

Although this console was marketed as a 32-bit system, the majority of its memory is only accessible through a 16-bit bus, meaning games will mostly use the Thumb instruction set to avoid spending two cycles per instruction fetch. Only in very exceptional circumstances (i.e. need to use instructions not found on Thumb while storing them in IWRAM), programmers will benefit from the ARM instruction set.

4.3 Becoming a Game Boy Color

Apart from the inclusion of GBC hardware (Sharp SM83, original BIOS, audio and video modes, compatible cartridge slot and so forth), there are two extra functions required to make backwards compatibility work.

From the hardware side, the console relies on switches to detect if a Game Boy or Game Boy Color cartridge is inserted. A **shape detector** in the cartridge slot effectively identifies the type of cartridge and allows the CPU to read its state. It is assumed that some component of CPU AGB reads that value and automatically powers off the hardware not needed in GBC mode.

From the software side, there is a special 16-bit register called `REG_DISPCNT` which can alter many properties of the display, but one of its bits sets the console to 'GBC mode' [7]. At first, I struggled to understand exactly when the GBA tries to update this register. Luckily, some developers helped to clarify this:

> I think what happens during GBC boot is that it checks the switch (readable at REG_WAITCNT 0x4000204), does the fade (a very fast fade, hard to notice), then finally switches to GBC mode (BIOS writes to REG_DISPCNT 0x4000000), stopping the ARM7.
>
> The only missing piece of the puzzle is what would happen if you were to remove a portion of the GBC cartridge shell so the switch isn't pressed anymore, then did a software mode-switch to GBC mode. Multi-boot mode could help here. I'm not sure if the switch needs to be pressed down for the GBC cartridge bus to work properly, or if it just works. I'm willing to guess that the switch is necessary for the bus to function, but that's just a guess.
>
> *– Dan Weiss (aka Dwedit, current maintainer of PocketNES and Goomba Color)*

5 GRAPHICS

Before we begin, you'll find the system a mix between the SNES and the Game Boy, the graphics core is still the well-known 2D engine called **PPU**. I recommend reading those articles before continuing since I'll be revisiting lots of previously-explained concepts.

Compared to previous Game Boys we now have a colour LCD screen that can display up to 32,768 colours (15-bit). It has a resolution of 240x160 pixels and a refresh rate of ~60Hz.

5.1 Organising the content

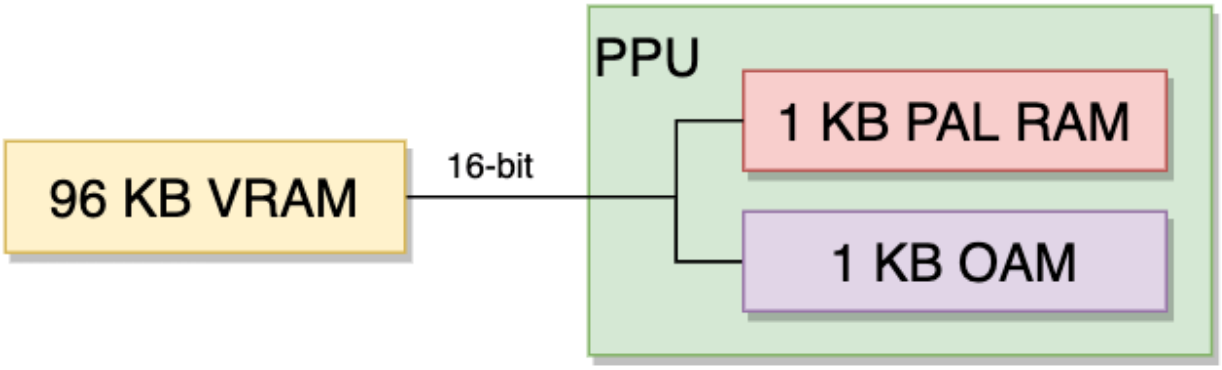

Figure 5.1: Memory architecture of the PPU.

We have the following regions of memory in which to distribute our graphics:

- 96 KB 16-bit **VRAM** (Video RAM): Where 64 KB store background graphics and 32 KB store sprite graphics.
- 1 KB 32-bit **OAM** (Object Attribute Memory): Stores up to 128 sprite entries (not the graphics, just the indices and attributes). Its bus is optimised for fast rendering.
- 1 KB 16-bit **PAL RAM** (Palette RAM): Stores two palettes, one for backgrounds and the other for sprites. Each palette contains 256 entries of 15-bit colours each, colour `0` being *transparent*.

5.2 Constructing the frame

If you've read the previous articles you'll find the GBA familiar, although there is additional functionality that may surprise you, and don't forget that this console runs on two AA batteries.

I'm going to borrow the graphics of Sega's *Sonic Advance 3* to show how a frame is composed.

5.2.1 Tiles

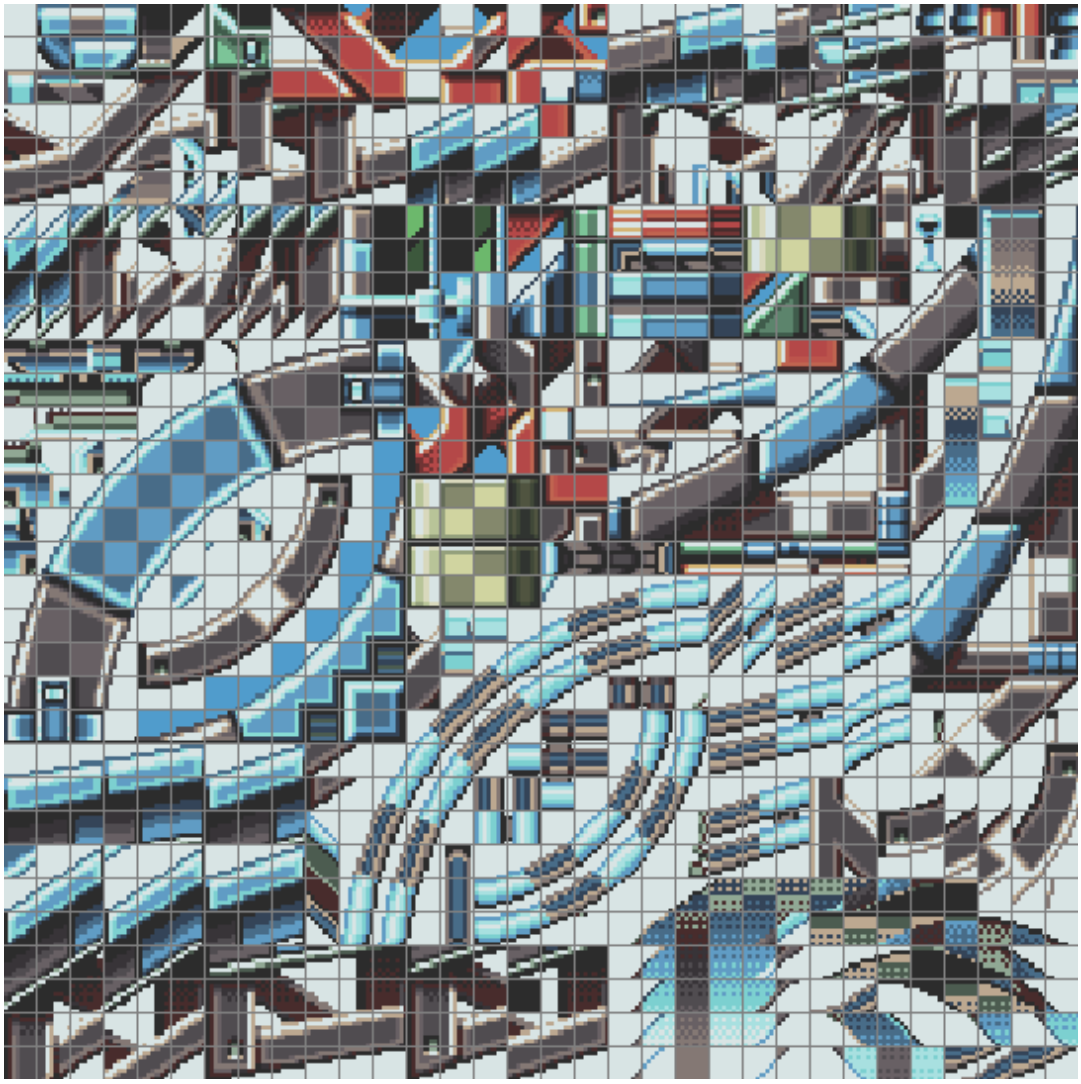

Figure 5.2: This block is made of 4bpp Tiles.

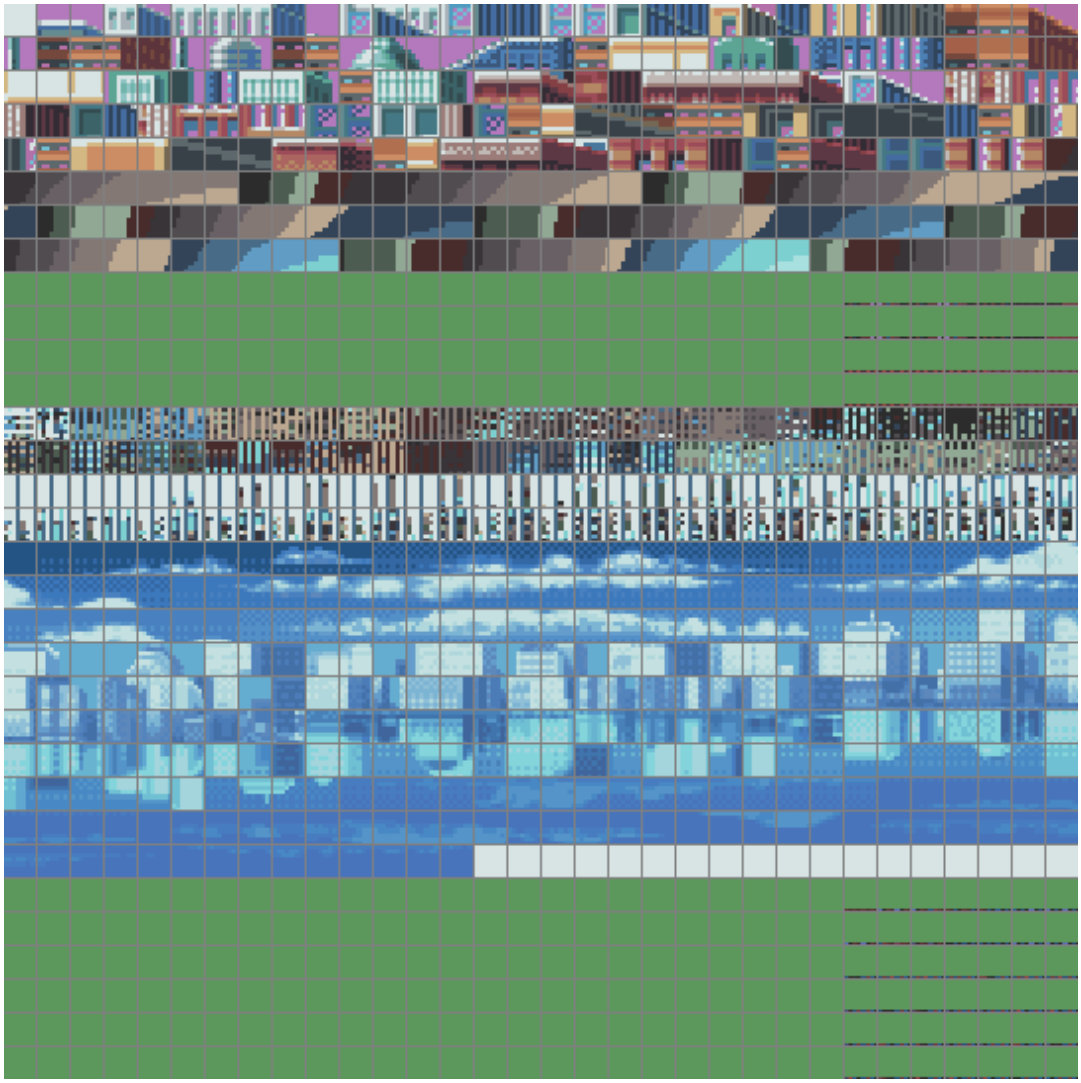

Figure 5.3: You may notice some weird vertical patterns in here, these are not graphics but ‘Tile Maps’ (see next section).

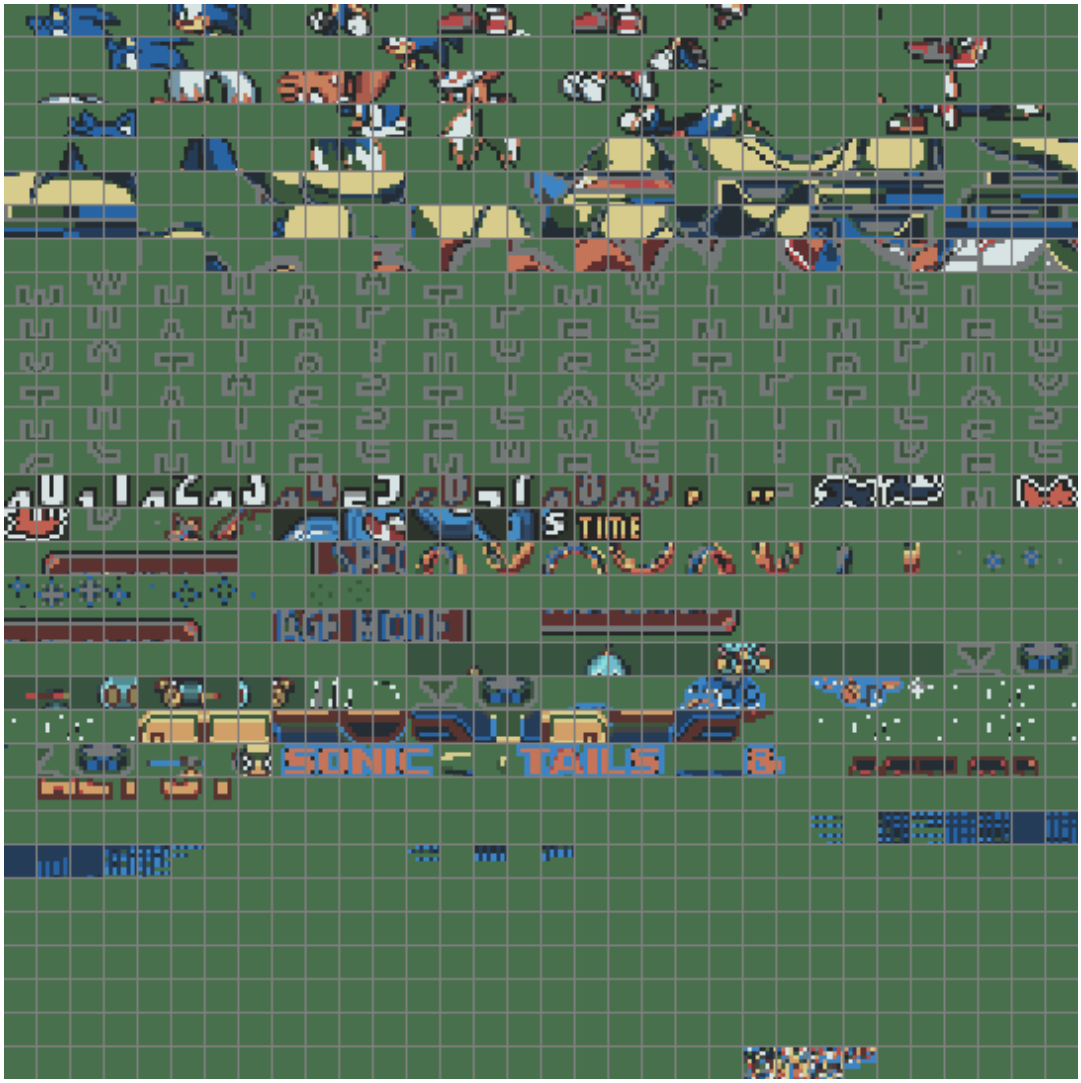

Figure 5.4: This block is reserved for sprites.

GBA's tiles are strictly 8x8 pixel bitmaps, they can use 16 colours (4bpp) or 256 colours (8bpp). 4bpp tiles consume 32 bytes, while 8bpp ones take 64 bytes.

Tiles can be stored anywhere in VRAM, however, the PPU wants them grouped into **charblocks**: A region of **16 KB**. Each block is reserved for a specific type of layer (background and sprites) and programmers decide where each charblock starts. This can result in some overlapping which, as a consequence, enables two charblocks to share the same tiles.

Due to the size of a charblock, up to 256 8bpp tiles or 512 4bpp tiles can be stored per block. Up to six charblocks are allowed, which combined require 96 KB of memory: The exact amount of VRAM this console has.

Only four charblocks can be used for backgrounds and two can be used for sprites.

5.2.2 Backgrounds

Figure 5.5: Background Layer 0 (BG0).

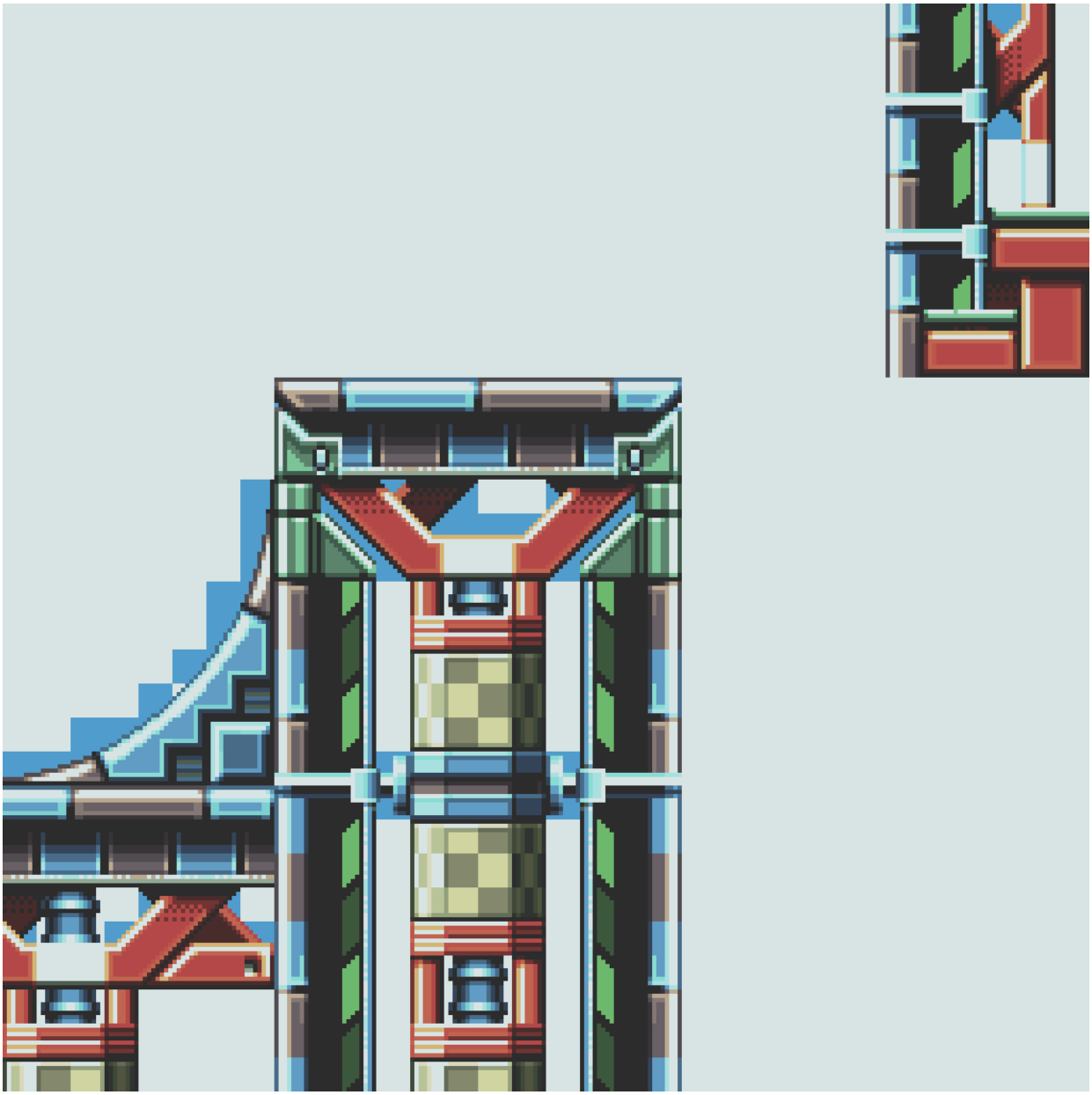

Figure 5.6: Background Layer 2 (BG2).

Figure 5.7: Background Layer 3 (BG3). This particular layer will be shifted horizontally at certain scan-lines to simulate water effects.

The background layer of this system has improved significantly since the Game Boy Color. It finally includes some features found in the Super Nintendo (remember the affine transformations?).

The PPU can draw up to four background layers. The capabilities of each one will depend on the selected mode of operation [8]:

- **Mode 0**: Provides four static layers.
- **Mode 1**: Only three layers are available, although one of them is **affine** (can be rotated and/or scaled).
- **Mode 2**: Supplies two affine layers.

Each layer has a dimension of up to 512x512 pixels. If it's an affine one then it will be up to 1024x1024 pixels.

The piece of data that defines the background layer is called **Tile Map**. To implement this in a way that the PPU understands it, programmers use **screenblocks**, a structure that defines portions of the background layer (32x32 tiles). A screenblock occupies just 2 KB, but more than one will be needed to construct the whole layer. Programmers may place them anywhere inside the background charblocks, this means that not all tiles entries will contain graphics!

5.2.3 Sprites

Figure 5.8: Rendered Sprite layer

The size of a sprite can be up to 64x64 pixels wide, yet for having such a small screen they will end up occupying a big part of it.

If that wasn't enough, the PPU can now apply **affine transformations** to sprites!

Sprite entries are 32-bit wide and their values can be divided into two groups:

- **Attributes**: Contains x/y position, h/v flipping, size, shape (square or rectangle), sprite type (affine or regular) and location of the first tile.
- **Affine data**: Only used if the sprite is affine, specify scaling and rotation.

5.2.4 Result

Figure 5.9: All layers merged (*Tada!*).

As always, the PPU will combine all layers automatically, but it's not over yet! The system has a couple of effects available to apply over these layers:

- **Mosaic**: Makes tiles look more *blocky*.

- **Alpha blending**: Combines colours of two overlapping layers resulting in transparency effects.
- **Windowing**: Divides the screen into two different *windows* where each one can have its own separate graphics and effects, the outer zone of both windows can also be rendered with tiles.

On the other side, to update the frame there are multiple options available:

- Command the **CPU**: The processor now has full access to VRAM whenever it wants. However, it can produce unwanted artefacts if it alters some data mid-frame, so waiting for VBlank/HBlank (*traditional way*) remains the safest option in most cases.
- Use the **DMA Controller**: DMA provides transfer rates ~10x faster and can be scheduled during VBlank and HBlank. This console provides 4 DMA channels (two reserved for sound, one for critical operations and the other for general purpose). Bear in mind that the controller will halt the CPU during the operation (although it may hardly notice it!).

5.3 Beyond Tiles

Sometimes we may want to compose a background from which the tile engine won't be able to draw all required graphics. Now, modern consoles addressed this by implementing a **frame-buffer** architecture but this is not possible when there's very little RAM… Well, the GBA happens to have 96 KB of VRAM which is enough to allocate a **bitmap** with the dimensions of our LCD screen.

The good news is that the PPU actually implemented this functionality by including three extra modes, these are called **bitmap modes** [9]:

- **Mode 3**: Allocates a single fully-coloured (8bpp) frame.
- **Mode 4**: Provides two frames with half the colours (4bpp) each.

- **Mode 5**: There’re two fully-coloured frames with half the size each (160x128 pixels).

The reason for having two bitmaps is to enable **page-flipping**: Drawing over a displayed bitmap can expose some weird artefacts during the process. If we instead manipulate another one then none of the glitches will be shown to the user. Once the second bitmap is finished the PPU can be updated to point to the second one, effectively swapping the displayed frame.

Figure 5.10: Super Monkey Ball Jr. (2002). Bitmap mode allowed the CPU to provide some rudimentary 3D graphics for the scenery. Foreground objects are sprites (separate layer).

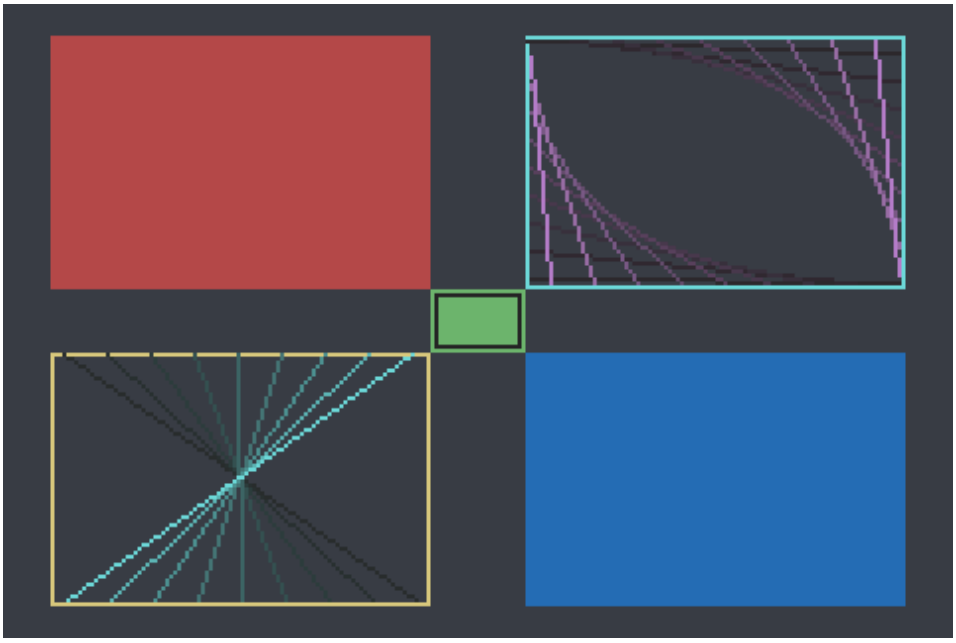

Figure 5.11: Tonc's demo. Rendered bitmap with some primitives. Notice the screen doesn't show significant patterns produced by tile engines.

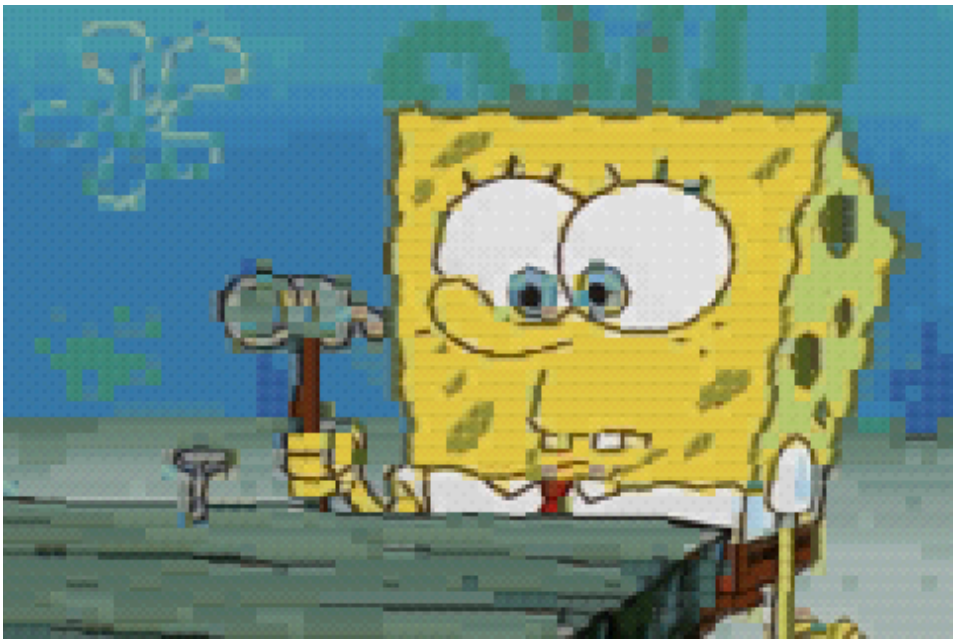

Figure 5.12: Nickelodeon's SpongeBob SquarePants. Episode distributed as a *GBA Video* cartridge (it suffered a lot of compression, of course).

Overall it sounds like a cutting-the-edge feature, however, most games held on to the tile engine. Why? Because in practice it **costs a lot of CPU**

resources.

You see, while using a tile engine the CPU can delegate most of the computations to the graphics chip. By contrast, the frame-buffer system that the PPU provides is limited to only displaying that segment of memory as a **single background layer**, which means no more individual affine transformations, layering or effects unless the CPU computes them. Also, the frame-buffer occupies 80 KB of memory, so only 16 KB (half) are available to store sprite tiles.

For this reason, these modes are used exceptionally, such as for playing motion video (**Game Boy Advance Video** completely relied on this) or rendering **3D geometry** with the CPU.

6 AUDIO

The GBA features a **2-channel sample player** which works in combination with the legacy Game Boy sound system.

6.1 Functionality

Here is a breakdown of each audio component using *Sonic Advance 2* as an example:

6.1.1 PCM

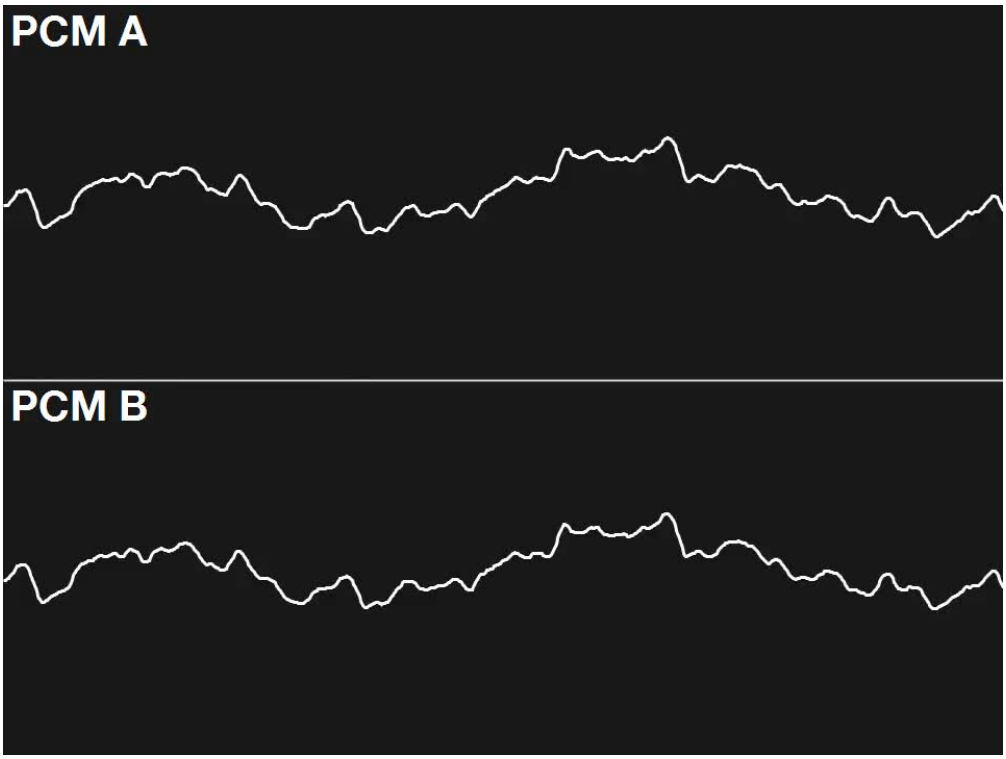

Figure 6.1: PCM-only channels. Video available in the website edition.

The new sound system can now play PCM samples, it provides two channels called **Direct Sound** where it receives samples using a **FIFO queue** (implemented as a 16-byte buffer).

Samples are **8-bit** and **signed** (encoded in values from -128 to 127). The default sampling rate is 32 kHz, although this depends on each game: since

a higher rate means a larger size and more CPU cycles, not every game will spend the same amount of resources to feed the audio chip.

DMA is essential to avoid clogging CPU cycles. **Timers** are also available to keep in-sync with the queue.

6.1.2 PSG

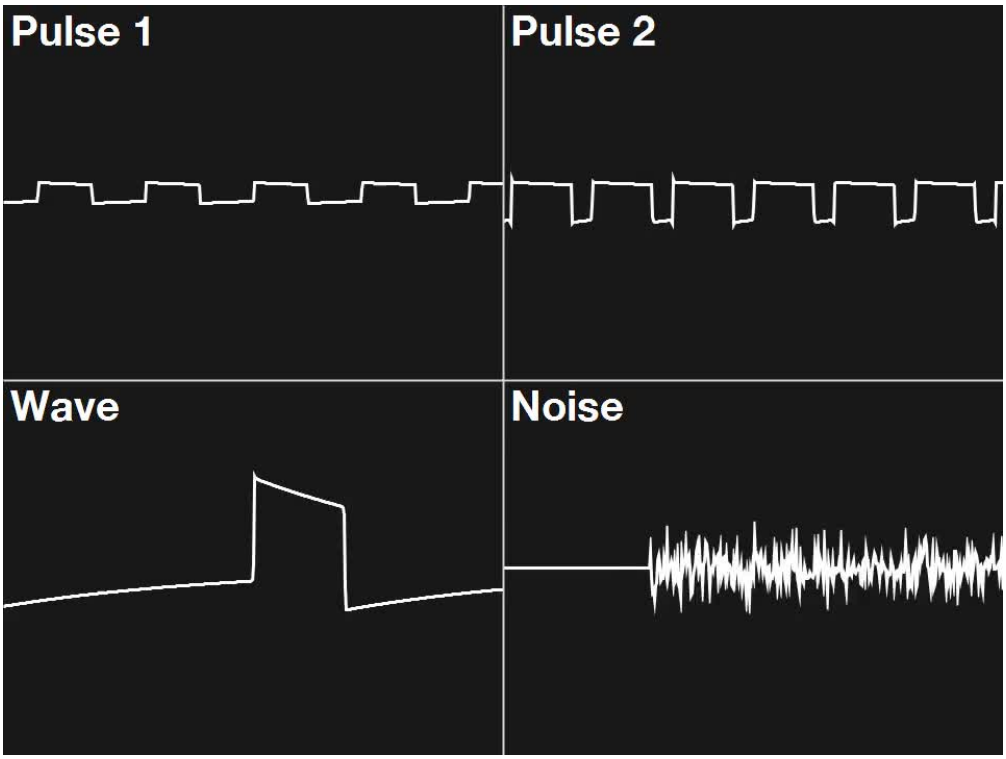

Figure 6.2: PSG-only channels. Video available in the website edition.

While the Game Boy subsystem won't share its CPU, it does give out access to its PSG. For compatibility reasons, this is the same design found on the original Game Boy. I've previously written this article that goes into detail about each channel in particular.

The majority of GBA games used it for accompaniment or effects. Later ones will optimise their music for PCM and leave the PSG unused.

6.1.3 Combined

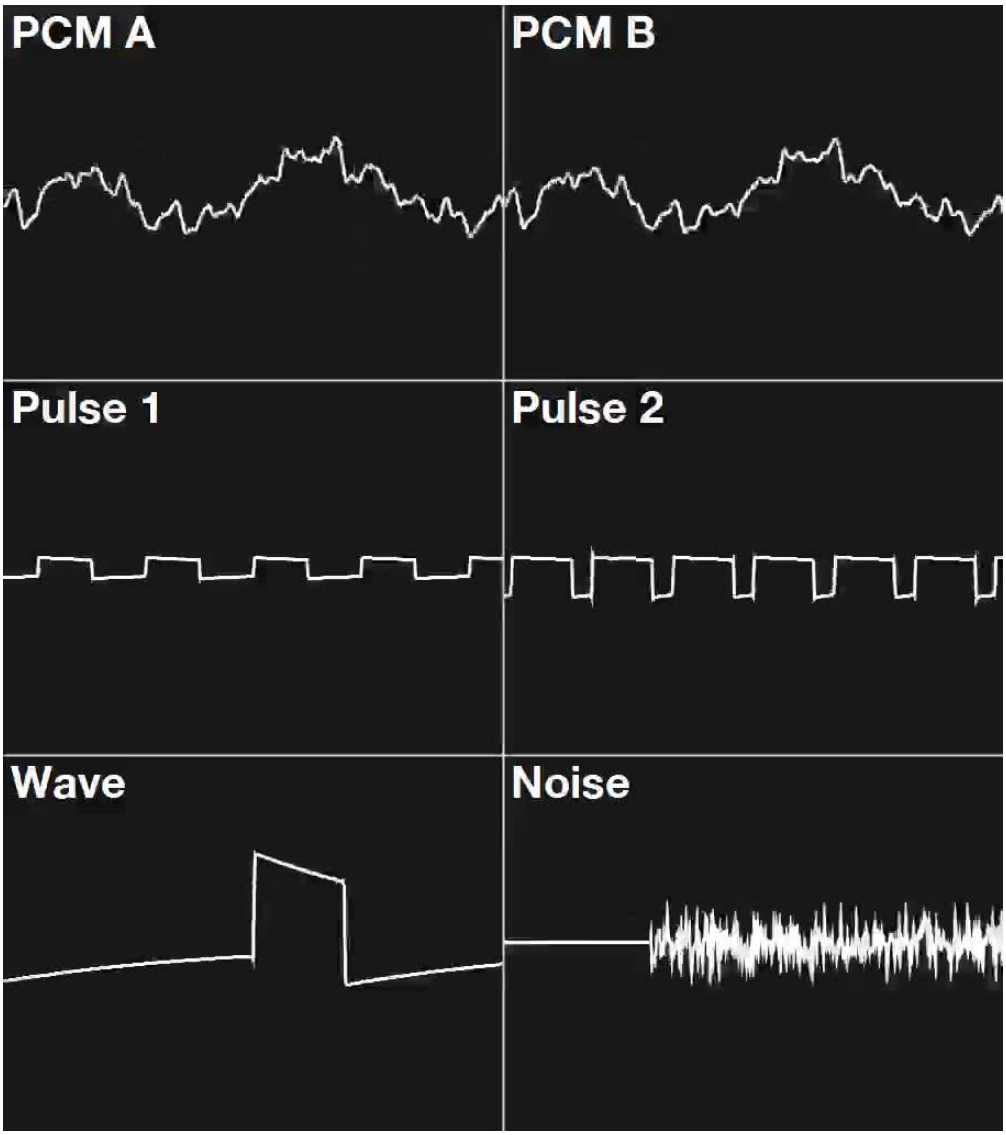

Figure 6.3: Tada! Video available in the website edition.

Finally, everything is automatically mixed together and output through the speaker/headphone jack.

Even though the GBA has just two PCM channels, some games can magically play more than two concurrent samples. How is this possible? Well, while only having two channels may seem a bit weak on paper, the main CPU can use some of its cycles to provide both audio sequencing and mixing [10] (that should give you an idea of how powerful the ARM7 is!).

Furthermore, in the ‘Operating System’ section, you’ll find out that the BIOS ROM included an audio sequencer!

6.2 Best of both worlds

Some games took the PCM-PSG duality further and ‘alternated’ the leading chip depending on the context.

In this game (*Mother 3*), the player can enter two different rooms, one *relatively normal* and the other with a *nostalgic* setting. Depending on the room the character is in, the same score will sound *modern-ish* or *8bit-ish*.

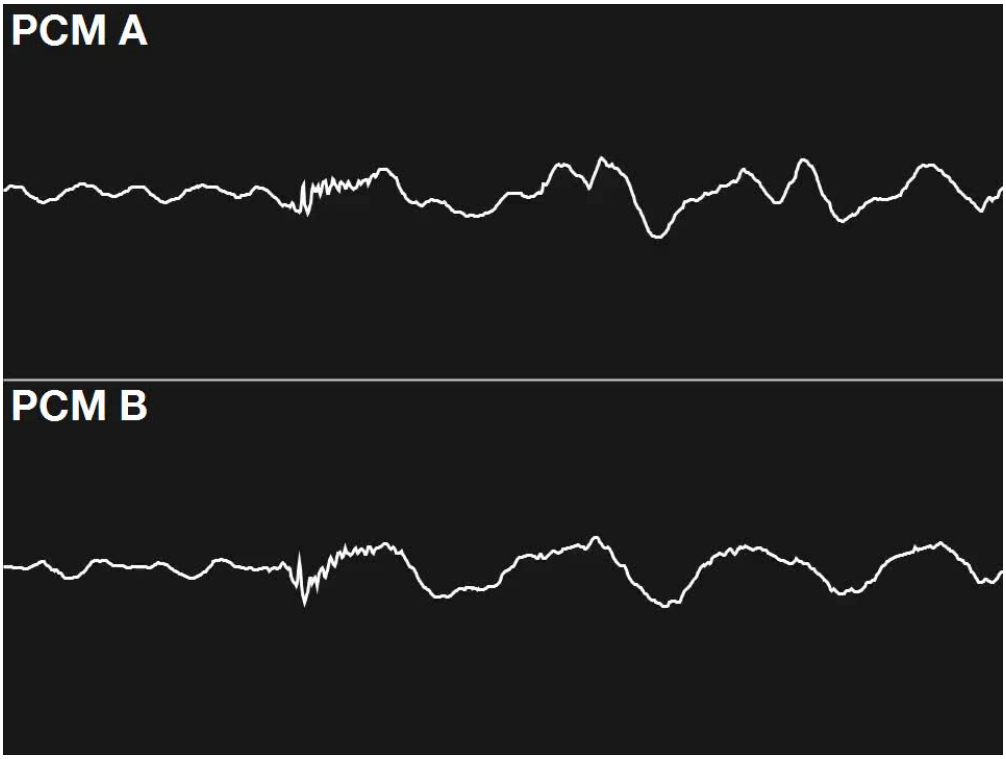

Figure 6.4: Normal room, only uses PCM. Video available in the website edition.

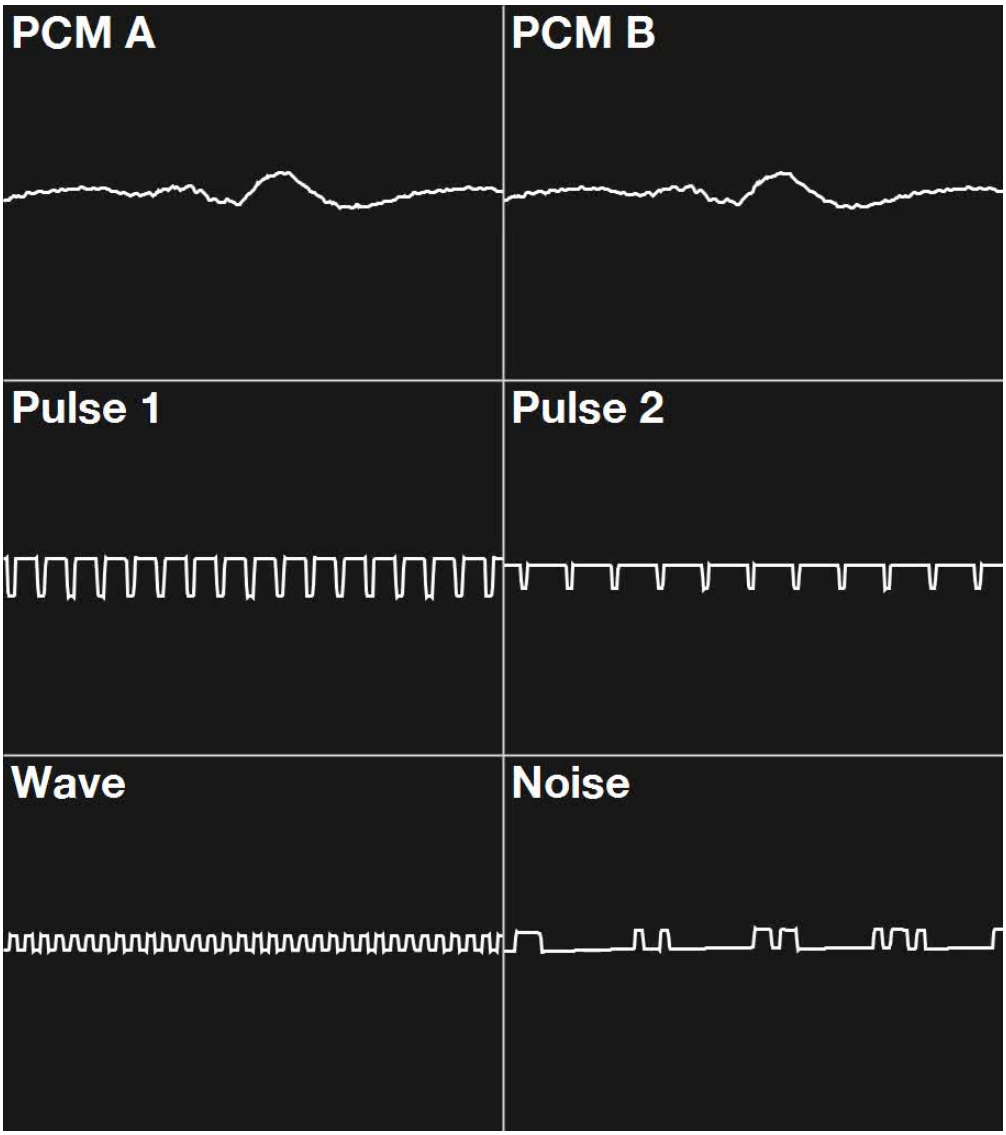

Figure 6.5: Nostalgic room, PSG leads the tune. Video available in the website edition.

7 OPERATING SYSTEM

ARM7's reset vector is at `0x00000000`, which points to a **16 KB BIOS ROM**. That means the Game Boy Advance first boots from the BIOS, which in turn shows the iconic splash screen and then decides whether to load the game or not.

That ROM also stores software routines that games may call to simplify certain operations and reduce cartridge size [11]. These include:

- **Arithmetic functions**: Routines to carry out Division, Square Root and Arc Tangent.
- **Affine matrix calculation**: Given a 'zoom' value and angle, it calculates the affine matrix that will be input to the PPU in order to scale/rotate a background or sprite.
    - There are two functions, one for sprites and the other for backgrounds. Their parameters are slightly different but the idea is the same.
- **Decompression functions**: Implements decompression algorithms including Run-Length, LZ77 and Huffman. It also provides bit unpacking and sequential difference.
- **Memory copy**: Two functions that move memory around. The first one copies 32-byte blocks using a specialised opcode for this type of transfer (`LDMIA` to load and `SDMIA` to store) only once. The second one copies 2-byte or 4-byte blocks using repeated `LDRH/STRH` or `LDMIA/STMIA` opcodes, respectively. Thus, the second function is more flexible but not as fast.
- **Sound**: Implements a complete MIDI sequencer! It includes many functions to control it.

- **Power interface**: Shortcuts for resetting, clearing most of the RAM, halting the CPU until a certain event hits (V-blank or custom one) or switching to ‘low-power mode’.
- **Multi-boot**: Uploads a program to another GBA and kickstarts it. More details are in the ‘Game’ section.

The BIOS is connected through a 32-bit bus and it’s implemented using a combination of Arm and Thumb instructions, though the latter is the most prominent.

Also, remember that all of this will only run on the ARM7. In other words, there isn’t any hardware acceleration available to speed up these operations. Hence, Nintendo provided all of this functionality through software.

8 GAMES

Programming for the GBA was similar to the SNES with the addition of all the advantages of developing games in the early 2000s: Standardised high-level languages, better compilers, faster RISC CPUs, non-proprietary computers for development, comparatively better documentation and… Internet access!

Programs are mostly written in C with performance-critical sections in assembly (ARM and Thumb) to save cycles. Nintendo provided an SDK with libraries and compilers.

Games are distributed in a new proprietary cartridge called **Game Pak**.

8.1 Accessing cartridge data

While the ARM7 has a 32-bit address bus, there are only 24 address lines connected to the cartridge. This should mean that up to 16 MB can be accessed on the cartridge without needing a mapper, however, the official docs state that **32 MB of cartridge data are mapped in memory**. So what's happening here? The truth is, the Gamepak uses **25-bit addresses** (which explains that 32 MB block) but its bottommost bit is fixed at zero, so the only 24 remaining bits are set. This is how Gamepak addressing works.

Now, does this mean that data located at odd addresses (with its least significant bit at '1') will be inaccessible? No, because the data bus is 16-bit: For every transfer, the CPU/DMA will fetch the located byte plus the next one, allowing to read both even and odd addresses. As you can see, this is just another work of engineering that makes full use of hardware capabilities while reducing costs.

8.2 Cartridge RAM space

To hold saves, Game Paks could either include [12]:

- **SRAM**: These need a battery to keep their content and can size up to 64 KB (although commercial games did not exceed 32 KB). It's accessed through the GBA's memory map.
- **Flash ROM**: Similar to SRAM without the need for a battery, can size up to 128 KB.
- **EEPROM**: These require a serial connection and can theoretically size up to anything (often found up to 8 KB).

8.3 Accessories

The famous **Game Boy Link Cable** provided multi-playing capabilities. Additionally, the cable has a special feature internally known as **Multi-boot**: Another console (either GBA or GameCube) can send a functional game to the receiver's EWRAM, then the latter would boot from there (instead of needing a cartridge).

9 ANTI-PIRACY & HOMEBREW

In general terms, the usage of proprietary cartridges was a big barrier compared to the constant cat-and-mouse game that other console manufacturers had to battle while using the CD-ROM.

To combat against *bootleg* cartridges (unauthorised reproductions), the GBA's BIOS incorporated the same boot process found in the original Game Boy.

9.1 Flashcarts

As solid-state storage became more affordable, a new type of cartridge appeared on the market. **Flashcarts** looked like ordinary Game Paks but had the addition of a re-writable memory or a card slot that enabled to run game ROMs. The concept is not new actually, developers have internally used similar tools to test their games on a real console (and manufacturers provided the hardware to enable this).

Earlier solutions included a burnable NOR Flash memory (not exceeding the 32 MB) and some battery-backed SRAM. To upload binaries to the cartridge, the cart came with a Link-to-USB cable that was used with a GBA and a PC running Windows XP. With the use of a proprietary flasher software and drivers, the computer uploaded a multi-boot program to the GBA, which in turn was used to transfer a game binary from the PC to the Flashcart inserted in the GBA. Overall, the whole task of uploading a game was deemed *sluggish*. Later Flashcarts (like the 'EZ-Flash') offered larger storage and the ability to be programmed without requiring the GBA as an intermediate [13]. The last ones relied on removable storage (SD, MiniSD, MicroSD or whatever).

Commercial availability of these cards proved to be a **grey area**: Nintendo condemned its usage due to enabling piracy whereas some users defended that it was the only method for running **Homebrew** (programs made outside game studios and consequently without the approval of Nintendo). Nintendo's argument was backed by the fact flashers like the EZ-Writer assisted users to patch game ROMs so they can run in EZ-Flash carts without problems. After Nintendo's legal attempts, these cartridges were banned in some countries (like in the UK). Nonetheless, they persisted worldwide.

10 THAT'S ALL FOLKS

Figure 10.1: My GBA and a couple of games. Too bad it doesn't have a backlit!

11 SUPPORT

If you enjoyed what you read and would like to help out, please consider donating. Your contribution will be used to fund the purchase of tools and resources that will help me to improve the quality of existing articles and upcoming ones.

You can go to the donation page to find out more.

Along with the donation you can leave a note stating which article in particular you want me to invest in. As a token of gratitude, your name will be included in the credits section of the next article or your nominated one, unless you state otherwise.

11.1 Other ways of contributing

Alternatively, you can help out by suggesting changes on the Github repository and/or adding translations on the Crowdin page.

11.2 Acknowledgments

I want to give a big thanks to the following people for their past donation since the website launched at the start of 2019:

- Adam Obenauf
- Adrian Burgess
- Alberto Cordeddu
- Alberto Massidda
- Alexander Perepechko
- Alí El wahsh
- Andreas Kotes
- Andrew Woods
- Antonio Bellotta
- Antonio Vivace

- BBQ Inc
- Ben Morris
- Bitmap Bureau
- Brenden Kromhout
- Carlos Díaz Navarro
- Chiang Yi-Ta
- Christopher Henderson
- Christopher Starke
- Cirilla Rose
- Colin Szechy
- Daniel Cassidy
- David Bradbury
- David Portillo
- David Sawatzke
- Dominic Wehrmann
- Dudeastic
- Duncan Bayne
- Dávid Major
- Eli Lipsitz
- Elizabeth Parks
- Eric Haskins
- eurasianwolf
- Gerald Mueller-Bruhnke
- Grady Haynes
- Guillermo Angeris
- Ifeanyi Oraelosi
- Izsak Barnette

- Jack Wakefield
- Jacob Almoyan
- Jaerder Sousa
- James Kilner
- James Knowler
- James Montgomerie
- James Osborne
- James William Jones
- Jan Straßenburg
- Jason Strothmann
- Jerre van der Hulst
- Joe Pugh
- Johannes Baiter
- John Mcgonagle
- Josh Enders
- Julius Hofmann
- ltlollo
- Luke Wren
- Manish Sinha
- Matthew Butch
- MCE
- Michael Chi
- Michal Borsuk
- Nathan Castle
- Neil Moore
- Nick T.
- Oleg Andreev

- Olivier Cahagne
- Owen Christensen
- Parker Thomas
- Paul Adamson
- Payam Ghoreishi
- petey893
- Phil Stevenson
- Piergiorgio Arrigoni
- Racri
- rocky
- Roger Bolsius
- Samuel Shieh
- Sanqui
- Shoaib Meenai
- Simon Pichette
- Stephen Molyneaux
- Stuart Hicks
- Sébastien Lethuaire
- Thomas Finch
- Thomas Lanner
- Thomas Peter Berntsen
- Tim Cox
- Ulrich Bogensperger

12 LICENSING

Just like the contents on the website, all of my writings and multimedia assets authored by me are available under a permissive Creative Commons license. This edition in particular, however, is available under the Creative Commons Attribution-NonCommercial 4.0 International (I had to make it ‘non-commercial’ in order to protect this document from being sold without my authorisation).

The chosen license also means you can use my content for your own work, what I ask is that you reference my work properly. Please pay attention when you quote or paraphrase my work, as you need to explicitly state when you do either. Unfortunately, throughout my time writing the articles, I’ve come upon derivative works that didn’t respect these simple steps (i.e. they don’t specify when they quote me or they don’t provide the complete URL of the article). To be fair, sometimes this is involuntarily but others are the result of lack of care, so I find myself dedicating more text explaining this. On the other side, there are derivative works which have surprised me (in the positive way) with their way of referencing my articles, providing beyond of what the CC license asks for.

Anyway, if you have any doubts, you can check this referencing guide made by the University of Leeds.

13 CHANGELOG

It’s always nice to keep a record of changes.

2021-05-18

- Renamed Gameboy CPU from ‘LR35902’ to ‘SM83’ (see Gameboy article’s changelog for reason).

2021-03-03

- Added ‘Operating System’ section with BIOS breakdown.

2021-01-06

- Added memory diagram.
- Big set of corrections and additions (see https://github.com/flipacholas/Architecture-of-consoles/issues/20), thanks @Dwedit, @selb and @AntonioND.
- Improved ‘Sources’ format.

2020-09-19

- Corrected grammar and added content, thanks @dpt
- Expanded section about gamepak’s 25-bit addressing

2020-08-22

- Clarified tile section

2019-12-09

- Even better Thumb explanation.
- Added more audio info.

2019-10-03

- Improved Thumb explanation

2019-09-17

- Added a quick introduction

2019-09-01

- Added my GBA 🧐

2019-08-26

- Used better wording on some explanations

2019-08-19

- Corrected wee mistakes

2019-08-18

- Ready for publication

14 SOURCES

The contents of this document could not have been possible without the research and expertise provided by the works listed here. Some are carried out for commercial purposes while others are entirely volunteer-driven.

[1] E. Amos, “The vanamo online game museum.” [Online]. Available: https://commons.wikimedia.org/wiki/User:Evan-Amos

[2] gbhwdb.gekkio.fi, “Game boy hardware database.” [Online]. Available: https://gbhwdb.gekkio.fi/consoles/agb/

[3] Arm Holdings, “ARM7TDMI (rev 3 core processor).” [Online]. Available: https://developer.arm.com/documentation/dvi0027/b/arm7tdmi

[4] D. Thomas, “ARM > introduction to ARM > thumb.” 2012 [Online]. Available: http://www.davespace.co.uk/arm/introduction-to-arm/

[5] J. Vijn, “GBA hardware - tonc v1.4.2.” 2013 [Online]. Available: https://www.coranac.com/tonc/text/hardware.htm

[6] V. Pfau, “Cycle counting, memory stalls, prefetch and other pitfalls.” 2014 [Online]. Available: https://mgba.io/2015/06/27/cycle-counting-prefetch/

[7] A. N. Díaz, “Gba-switch-to-gbc.” Github [Online]. Available: https://github.com/AntonioND/gba-switch-to-gbc

[8] I. Finlayson, “GBA tile modes.” University of Mary Washington [Online]. Available: https://web.archive.org/web/20190516091420/http://ianfinlayson.net/class/cpsc305/notes/13-tiles

[9] I. Finlayson, “GBA bitmap modes.” University of Mary Washington [Online]. Available:

https://web.archive.org/web/20190515050420/http://ianfinlayson.net/class/cpsc305/notes/09-graphics

[10] C. Strickland, “Audio programming on the GameBoy advance.” 2002 [Online]. Available: https://www.gamedev.net/articles/programming/general-and-gameplay-programming/audio-programming-on-the-gameboy-advance-part-1-r1823/

[11] J. Vijn, “BIOS calls.” 2013 [Online]. Available: https://www.coranac.com/tonc/text/swi.htm

[12] R. Ziegler, “GBA ROM cartridge interface.” 2008 [Online]. Available: http://reinerziegler.de.mirrors.gg8.se/GBA/gba.htm

[13] ezfadvance.com, “EZ-flash 2.” [Online]. Available: https://web.archive.org/web/20200216121318/http://www.ezfadvance.com/cards/EZ-Flash_2.htm