# ZX Spectrum (48K) in a FPGA (and a CPC-464 too)

Jesús Arias

## 1 Introduction

The way to go computing here in Spain during the eighties was with a ZX-Spectrum, or, if you could afford it, with an Amstrad-CPC. Apple-twos and Commodores were for dollar-filled americans with NTSC monitors, and anything else was just a curiosity. So, it's no wonder a lot of retro revival is now focused on these systems, specially to the Spectrum due mainly to its sheer simplicity.

Sir Clive Sinclair, and Allan Shugart were the visionaries behind these computers and they surely deserve a recognition as high as Steve Jobs or Bill Gates, but this is not America, you know. And anyway, all these people were just that: visionaries. The actual people who designed these computers were engineers barely known if known at all (Wozniak is the only engineering star I can recall). These people had to work under the pressure of their visionary bosses who were always demanding the impossible for yesterday, and they did what they did. These computers were quite remarkable, but surely they could have been improved.

Looking at any Sinclair product today they all can be categorized as a good-looking, desirable, easy-to-break, crap. Their designers cut so many corners that they ended reinventing the hula-hop. And in that respect the ZX Spectrum is without doubt the highest quality product ever coming out of the Sinclair factories, something the owners of the Sinclair's electric vehicle surely recognize.

The Amstrad line of products feared better, but they were still paces away from, let say, equivalent Sony products (Not to mention Amstrad Hi-Fis). But digging under their surface you can see these products weren't the vanguard of the technology of their time. On the contrary, they were always years behind their competitors. Yet, they managed to win a considerable market share thanks to the CPC computer, and then Amstrad slowly faded away with the PCW typewriters and PC clones.

So let's start dissecting Spectrums and CPCs. The ZX spectrum was a hit because it offered a Z80 computer with a decent amount of memory for an affordable price, something their predecessors, the ZX-80 and ZX-81, didn't, and this was enough to persuade buyers to forget about the many inconveniences of the Spectrum, like...

- An horrible rubber keyboard that lasted no more than a thousand lines of code.
- An equally horrible RF modulator for the video signal. Yes, many TV's of that time didn't have a video input, so the computer had to be attached to the antenna jack. But the SCART connector was also starting to become common for new PAL TVs.
- A sadistic BASIC keyword input.
- Unreliable tape storage. It was not only slow. Only a weak checksum was added at the end of the data, so you had to wait until the tape stop making noises to realize something was wrong.

But first let's take a look to the electronics inside the case. One may ask what defines the hardware of a particular computer, and in the Spectrum case it is without doubt the Uncommitted Logic Array. The rest is mainly the Z80 itself, the memory, the power supply, and little more. So let's talk a little about the ULA.

## 1.1 Committing the Uncommitted Logic Array

Uncommitted Logic Array is a very descriptive name given by Ferranti to its line of Gate Arrays (other silicon manufacturers opted to name these kind of chips Gate Arrays). ULAs or GAs were chips with a lot of transistors arranged in a regular two-dimensional array. These chips lacked the final metal layer (in these days the ICs only had one metal layer), so they were not more than a collection of unconnected transistors, or in other words: they were "uncommitted". These chips were kept in wafers and stored until some customer, let say Mr Clive, come asking for the design of some digital chip for this or that computer he had on is mind. The Ferranti engineers then proceeded to design a metal layer for the ULA that matched the customer schematic and some wafers then went again to the factory line to get the metal layer that "committed" them to become a video chip for the new ZX Spectrum.

By reducing the design of a custom chip to a single metal layer the cost of the chip was also much more affordable. At least if many thousands of them were to be ordered, this approach still was very expensive for small runs. And, of course, any mistake would be enough to ruin not only your budget, but also to delay your product by months or years. That was the main motivation for the development of modern day Field Programmable Gate Arrays. In these modern chips the connection between subcircuits isn't made with metal lines, but with multiplexers already built into the chip and controlled by digital data. But the basic idea is still the same: The FPGA is uncommitted until you upload some bitstream into its configuration memory, and then it performs the function you like.

But, if someone mention you an ULA you can be sure he's talking about a circuit committed to perform the duty of a video controller in a ZX Spectrum. The ULA of this computer was the best seller product of Ferranti, and for may of us the only known Ferranti product (I think that company also worked for the military, but when the Spectrum computer was over Ferranti went to the bottom with it)

The circuit inside the Spectrum ULA isn't very complex (In fact a modern Spectrum clone, the ZX Harlequin, is built using TTL chips instead). It basically includes:

- The horizontal and vertical counters for the generation of PAL video synchronism.
- The generation of video memory addresses, both for pixels and for video attributes (colors). These addresses are just the concatenation of some horizontal and vertical counter bits.
- The shifting of video data and the multiplexing of foreground and background colors.
- The arbitration of memory contention between itself and the Z80. This if achieved by forcing the CPU clock high during the time it takes to perform the video memory read by the ULA. In the Spectrum PCB resistors were placed in series with the Z80 address and data buses in order to save multiplexers: If the ULA and the Z80 contend for the memory the ULA is going to win because of its lower output impedance. This can be described either as a very clever trick or as a heap of shit due to the signal degradation these resistors introduce (I'm biased towards the second opinion).
- A single output port for setting the color of the border of the screen and for audio and cassette outputs.
- A single input port for keyboard scanning and for cassette input. The I/O port is accessed for any I/O address with the lower bit low. Again, electrical contentions can happen if an IN instruction is executed with more than one address bit low if there are expansion boards attached. Why not to decode the 8 low address bits of the bus instead of just A0 is something that puzzles me. So cramped was the ULA that a simple 8-input AND gate couldn't be fit?

But the main criticism I got about the ULA design is the way memory contention is managed, because when the ULA is displaying video the Z80 is going to be stopped very often (on average 5 out of 8 cycles). But the Z80 already include a /WAIT signal that can force it to postpone its read or write operation until the video logic finishes its read. The use of the /WAIT input would require a more deep knowledge of the Z80 timings, but it will result in far less lost cycles than in the Spectrum ULA. BTW, this is how the memory contention is managed in the gate array of the Amstrad-CPC, where it results in one cycle penalty for 3-cycle reads and writes and no penalty for 4-cycle accesses like op-code fetches and I/O.

But if the GA of the Amstrad-CPC looks hi-tech take into consideration the one of the Inves Spectrum+, a clone of the Spectrum 48K made by Investronica, an Spanish company. In the Inves Spectrum+ there is no memory contention at all. Those guys at Investronica managed to run the Z80 without any wait state while also displaying video! That's top quality design! Also in that clone you won't find anything like the resistive multiplexing crap. Its a pity that such fine work was done for the clone of an aging computer. Not only that, the Inves Spectrum+ got a bad reputation due mainly to some games with compatibility issues because they relied on the flawed behavior of the original Spectrum, and also to some negative reviews in computer magazines. They went as far as stating that the Inves computer could be damaged if certain code was executed on it. A fake news of the eighties that ruined the reputation of a clone far better than the original.

"The ZX Spectrum ULA How to Design a Microcomputer" is an interesting book I read after the design of my FPGA replica of the Spectrum. It includes a lot of information about the internals of this chip and its many flaws, some of them deserving at least a comment. In my opinion the more serious are:

- The composite sync generated by the ULA during vertical retraces doesn't follow the PAL standard at all. It is only a single pulse. Even with this many TVs of the era were able to show an stable image. I'm not sure if the same happens with more modern TVs. Mr Clive surely was super happy with this simplification.

- During the vertical border of the image the 16KB DRAM stops refreshing. This time is 7.68ms while the DRAM chips specs states a maximum refresh period of 2ms. I wonder how many DRAM chips were labeled as defective due to this. This is a first order engineering blunder.

- Blinking characters leave vertical lines on the screen when off. This is due to combinational glitches and could have been fixed by registering the GRB outputs using flip-flops.

- The refresh pulses of the Z80 interfere with video memory reads in the ULA and causes an "snow effect" even when the 16KB DRAM never gets refreshed by the Z80.

What more can be said about the ULA? Only that it is a power inefficient and delicate chip. Its core logic is bipolar and runs with a 0.95V power supply. Internal voltage regulators are included in order to lower the 5V of the supply pin to the 0.95V of the core and a lot of electrical power is converted into heat in them (to think that the power dissipation of the ULA could have been reduced to 1/5 of its value by simply providing a separate power pin for its core...). Also, the ULA can be easily damaged. The usual procedure for the destruction of an ZX Spectrum involves the connection of an expansion board, like a Kempston joystick, with the computer already powered. The most probable victims are the ULA and the ROM. The Z80 seems to be hard to kill, and that's remarkable because between its pins and the expansion connector there are nothing but copper traces. The ULA probably gets broken just because it's replacements are hard to find, while a more scientific explanation would be the latch-up problem where some parasitic transistors are turned on by a voltage transient and they short the power supply to ground. One should expect a more robust chip from a company that also worked for the military, but I'm starting to think Ferranti won the contracts just because it was British. Well, apart from the Spectrum ULA the only other barely known Ferranti chip was a shitty 3-pin AM radio receiver and almost all competitors were using CMOS at that time.

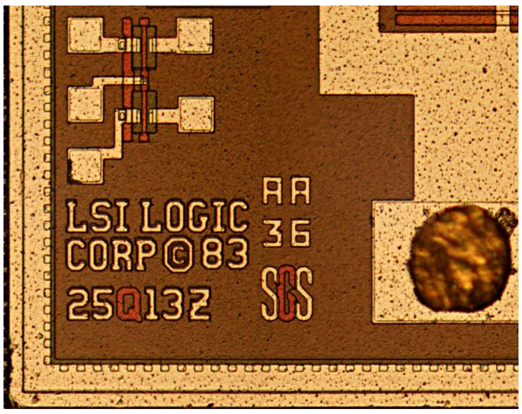


Figure 1: Detail of an Amstrad Gate Array chip

### 1.2 And what about Gate Arrays?

Well, in fact the TAHC10 of the already mentioned Inves Spectrum+ was a gate array made by Texas Instruments, and it was a CMOS chip, not a bipolar one like Ferranti's ULAs. But the Gate Array I want to mention now is that of the Amstrad-CPC. In this case it is difficult to point to a particular manufacturer because the chip comes with Amstrad markings (400xx), but a microphotograph published in the CPCWiki page shows what looks like a CMOS chip made by LSI Logic and/or SGS. Well, a more extensive search also found that older GA (40007 and 40008) were in fact bipolar ULAs made by Ferranti, while "modern" 40010s were the mentioned CMOS gate arrays.

When compared to the Spectrum ULA the first thing you notice is the GA does less things than the ULA because the CPC in addition to the GA also includes a 6845 CRT controller. What is doing a 6800 peripheral in a Z80 system like the CPC? Well, it is in charge of video timing and video address generation. It is fully programmable, allowing both PAL and NTSC timings, but this makes little sense in a computer sold along its own video monitor. The GA is thus in charge of video reading and shifting, memory contention management, palette storage, and little else. Some smart guy at Amstrad tough about using tristate outputs for the R, G, and B color outputs obtaining a 27-color palette instead of a 8-color one, and much of the GA complexity is related to this palette. But in overall the CPC GA seems to have little logic inside, and one starts to think why they didn't get rid of the 6845 by integrating a simplified version of it inside the GA.

There is no easy answer for that. Maybe the gate array wasn't big enough for all the logic, yet the Inves GA was capable enough of something like this. Or maybe the designers of the CPC were too conservative and resorted to a well known CRT controller, keeping the GA functions to a minimum in order to reduce the chances for mistakes. This last possibility is also supported by other curious schematic clues. For instance the sound chip, the AY-3-8912 PSG, isn't connected to the Z80 data bus directly. They resorted to drive the PSG bus from two ports of an 8255 PPI and to do some bitbanging for accessing the sound chip in the software.

Putting the PSG in the main data bus would have freed 10 pins of the 8255 at the cost of a few gates for the generation of the two control signals of the PSG. Well, this is already done in the Spectrum-128 and many other computers, so it wouldn't have been so difficult. And by the way, in the board they already have an underused gate array. So, why it is done this way?

Another probable answer is that the CPC board was designed in a rush and anything that worked in a breadboard was translated to the schematic without double thinking. "If it works keep it like that. Don't touch anything" may have been the order from above...

And BTW, the CPC also has the same incomplete I/O decoding we found in the Spectrum. This starts to look like a British tradition...

So the very revered CPC hides under its case a crappy design that ended converted into a standard due to the required compatibility with older systems. The 6845 and the PSG to PPI connection were present in all CPC models. Anyway, these ugly hardware details were hidden from the user via “warranty void if broken” labels and also by what was probably best BASIC ROM of all the computers of the time, even capable to put the GWBASIC of PCs to shame. The 6845 programmability was also exploited to achieve a fast text scrooling by means of video base address changes, and in overall the CPC was a very decent computer with a fast BASIC, a perfect video quality thanks to its RGB monitor, a good quality keyboard, and a cassette integrated into the case later replaced with a floppy disk in high-end models. Why Mr. Shugart choose an incompatible floppy format is another mystery, but I think the 3-inch floppy, and also the QL microdrive, were the collateral damage of too much success in the market. “Everybody is going to go my way” may Clive and Allan have tough, fully intoxicated with money. Yet, this really happened with the IBM PC and with Intel USB, and dreams come for free.

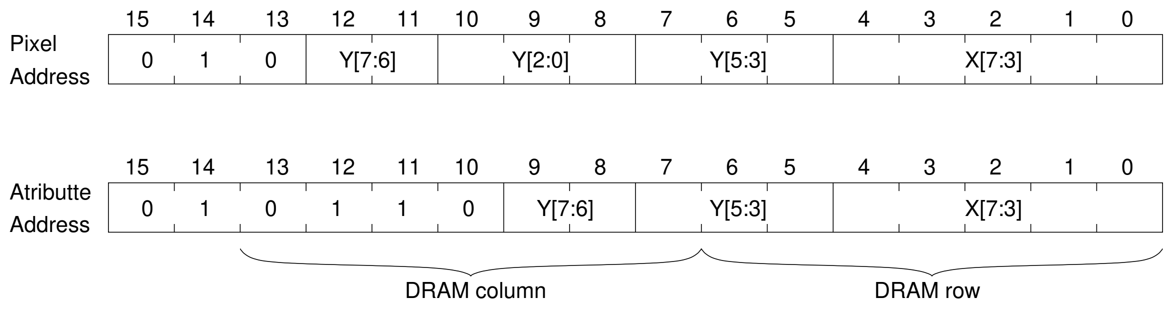
## 1.3 The ZX Spectrum 48K

The ULA of the ZX Spectrum provides a single video mode with 256×192 resolution and 1 bit per pixel. In addition to this, there is a 32×24 attribute array where the background and foreground colors of the corresponding 8×8 pixel block are stored, along with an intensity bit that makes the colors more intense when one, and a blink bit that makes the background and foreground colors to toggle every 320ms when one. These attributes seems to have been inspired by Teletext and they ended being a nightmare for game developers because they are the cause of the well known “attribute clash” problem. The contents of each attribute byte are:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| Blink | Intense | GRB bg | | | GRB fg | | |

The three bits in the background and foreground color fields have the green component as the MSB and the blue component as the LSB. The idea behind this order was to achieve an increasing 8-level gray scale on black and white TVs (higher values always more intense than lower ones)

The video memory starts at address 0x4000 and its size is 6 KB for the pixel array plus 768 bytes, starting at 0x5800, for the attribute array. But the mapping between pixels and memory isn’t an straight one. First, the screen is divided in three areas, each of 256×64 pixels wide, starting at addresses 0x4000, 0x4800, and 0x5000. In these areas each byte contains 8 pixels, with the MSB being displayed on the left. Increasing the address by one the position on the screen advances 8 pixels to the right until the end of the line is reached. But then 7 lines are skipped, so, the address 0x4020 corresponds to the beginning of the line #8 of the first area. Only when the address in increased by 256 we move to the next line. In summary, the video addresses are generated from the horizontal and vertical counters in the following way:



Where X and Y are the binary values of the counters. Notice that the 3 lower bits of X aren’t included because each byte of memory stores 8 pixels, and that the 8 lower bits are the same for both the pixel and attribute addresses. It seems that this strange mapping was selected in order to to exploit the page mode read of the dynamic RAMs of the Spectrum (same memory row for pixel and attributes).

Apart from the video, the ZX Spectrum is a very simple computer, with only one I/O port used for keyboard scanning and for the audio and cassette interface. This port is in fact two, one for reading and another for writing, and both ports are built inside the ULA. The output port also includes 3 bits for the selection of the color of the border of the screen. Interestingly, there are no output bits for the selection of the keyboard row. The high 8 bits of the address bus of the Z80 are used for this purpose instead. These ports are selected when an IN or OUT instruction is executed by the Z80 and the lower address bit, A0, is low. No I/O contentions were considered at all, so it is mandatory to have all the other address bits high to avoid them. The resulting I/O address is then 0xFE (the Z80 IN and OUT instructions use actually 16 bit addresses but the high 8 bits are only used when reading the keyboard) The bit fields of the input and output ports are:

Output port

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| - | - | - | Speaker | Cassette out | GRB border | | |

Input port

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| - | Cassette in | - | Keyboard row | | | | |

The ULA was short of pins and the cassette and speaker signals were all passed through a single pin. The designers were very proud of this and they even patented what, in retrospect, can be described as a crappy solution. The main idea was to generate a higher amplitude for the speaker pulses that for the cassette output. Two silicon diodes in series with the speaker then muted the cassette signal but allowed the higher pulses to reach the speaker. Well, you aren't supposed to apply a DC current to a speaker, to start with, so a diode in series to a speaker is something that causes nausea to any decent audio electronics engineer. Yet, this crap was patented.

With this knowledge about the ZX Spectrum internals we can start the design of our FPGA replica. We still have to know what keys are present on each keyboard row, that are:

| IN address | row | col 4 | col 3 | col 2 | col 1 | col 0 |
|---|---|---|---|---|---|---|
| 0xFEFE | 0 | V | C | X | Z | Caps Shift |
| 0xFDFE | 1 | G | F | D | S | A |
| 0xFBFE | 2 | T | R | E | W | Q |
| 0xF7FE | 3 | 5 | 4 | 3 | 2 | 1 |
| 0xEFFE | 4 | 6 | 7 | 8 | 9 | 0 |
| 0xDFFE | 5 | Y | U | I | O | P |
| 0xBFFE | 6 | H | J | K | L | Enter |
| 0x7FFE | 7 | B | N | M | Symbol Shift | Space |

So lets start with the design of a compatible VGA version of the video controller. After achieving this, the rest of the computer is little more than the CPU and the memory.

## 2 Designing a Ghost-ULA

A project I tough about but I never built was an adapter for watching the Spectrum screen in a VGA monitor. The VGA timing is twice as fast as a PAL video signal and all the video electronics inside the Spectrum was useless. The only way I found to achieve this was to build an entire video controller for the VGA and to attach it

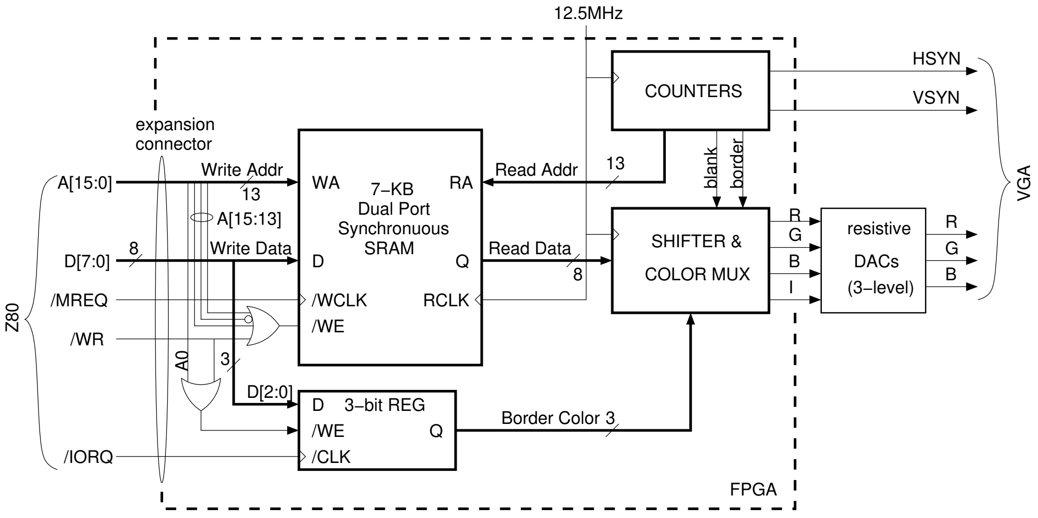


Figure 2: Block diagram of the Ghost-ULA

to the expansion connector of the Spectrum. This circuit was too complex to build using TTLs, and therefore I tough about using an FPGA for this purpose. Interestingly, the FPGA I knew included a dual-port memory that was very well suited for this application as it allows to perform independent read and writes simultaneously. That means no worries about memory contentions, you can read and write the RAM at any rate you like. The only problem here is the small amount of memory of the FPGA, but the ZX video memory takes only 6,75KB and that was less than the available RAM in the FPGA.

The block diagram of the possible VGA adapter is shown in figure 2. I named it Ghost-ULA because it performs a similar role as the Spectrum's ULA but it is completely unnoticed by the computer. Any Z80 write to the video memory is also done to the GULA RAM, so, after an initial screen erase the contents of the Spectrum video memory and the FPGA memory are the same, and the image on the VGA screen is going to be the same as the one on the TV, but with a much better quality.

The GULA is a write-only device with addresses decoded to enable writings when the processor stores a data in the \$4000 to \$5BFF address range. But it would be easy to map the location of the GULA memory to any address multiple of 8KB, for instance by using 3 dip-switches, and to have two different screens, one in the TV and another in the VGA monitor. The GULA RAM can even be mapped to ROM addresses (\$0000 or \$2000), so no extra memory would be required for the VGA display, the only limitation being the impossibility of reading back the video data.

The GULA idea never ended becoming a board because the TV RF signal wasn't the only thing I disliked about my ZX Spectrum. Its keyboard membranes were also broken and I didn't want to replace them with the same crappy stuff. So I started thinking about building the entire computer inside the FPGA. When a found a Z80 core source in Verilog and I could get a decent estimate of the CPU size measured in logic cells, the new project started to take shape. And, of course, it will include the Ghost-ULA as its video controller.

## 3 The ZX++, An Spectrum with only Three chips.

I mean: one FPGA, one SRAM, and one SPI Flash to store the configuration of the FPGA and the Spectrum ROM. Well, some other chips were also required, like two voltage regulators. That was the IC list of a board built around an ICE40HX4K FPGA. This chip is sold as a 3520 logic cell device, but the truth is it actually have 7680 logic cells. More than enough to fit a Z80 that requires about 2200 logic cells. This board was designed for the implementation of a range of 8-bits computers, and includes:

- One ICE40HX4K FPGA with 7680 logic cells and a total 16KB of dual-port synchronous SRAM.

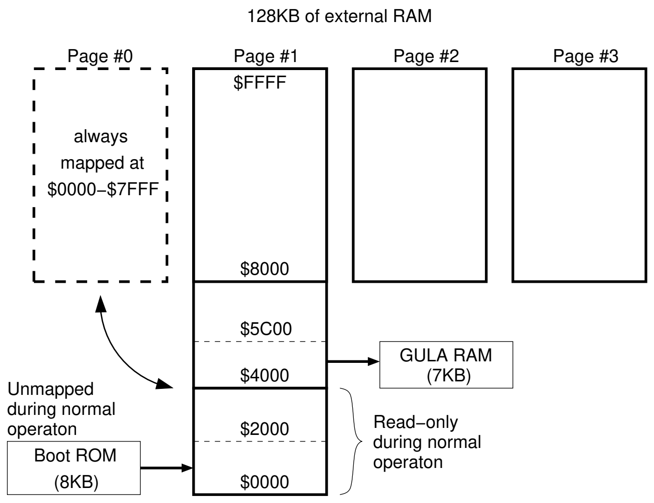


Figure 3: Memory map of the ZX++

- One 64K by 16-bit SRAM with an additional 128KB of memory.
- One N25Q64 SPI Flash memory with 8MB of nonvolatile storage.
- An SD card connector.
- A 4096 color VGA DAC, 4 bits or 16 levels for each color component. The three DACs are built using resistors.
- A PS2 keyboard connector.
- A joystick connector.
- A single bit amplified audio output. Capable of Pulse Width Modulation.

A description of the ZX++ designed for this board follows.

## 3.1 Memory map and I/O

In the board we have enough memory for the implementation of an Spectrum 48K. The main idea is to use the lower 64KB of the external SRAM for its 16KB of ROM and 48KB of RAM, while the internal memory of the FPGA is used for the GULA and an additional boot ROM. This last memory is mapped to address $0000 at startup and the main purpose of its code is to fill the first 16KB of the RAM with the code of the Spectrum's ROM. This is easy to do because during startup the lower 16KB of the memory can be written. Also, during startup there are more hardware resources available, in particular:

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| OUT, IN ($FF) | MOSI | SCK | $\overline{\text{CS1}}$ | $\overline{\text{CS2}}$ | BOOT | $\overline{\text{PG1}}$ | PG0 | MISO |
| | R/W | R/W | R/W | R/W | R/W | R/W | R/W | RO |

- An input/output register at IO address $FF with the SPI bus outputs (MOSI, SCK, /CS1 (SPI Flash), and /CS2 (SD card)) and three control bits: BOOT and Page selection. All output bits in this register start as high after reset, so, BOOT is enabled and while this bit is one we can write to this register. After the BOOT bit is written with zero this register is no longer accessible, the boot ROM is disabled, and the first 16KB of the memory are set as read-only. The BOOT bit also has the effect of disabling the clock divider when one, so, the Z80 runs at full speed (25MHz) until the BOOT is written with zero, and then, it runs at 1/7 of the former speed (3.57MHz). There are also two control bits related to memory banking: /PG1 and PG0. These bits select which page is mapped to the $8000 to $FFFF address range. PG1 is inverted, and therefore, we have the following mapping:

| PG[1:0] | Page at $8000-$FFFF | Comments |
|---|---|---|
| 11 | 1 | Normal selection for ZX emulation |
| 10 | 0 | Same memory as $0000-$7FFF. Not used |
| 01 | 3 | Highest memory page (for TAP file storage) |
| 00 | 2 | High memory page (for TAP file storage) |

Pages #2 and #3 aren't used during normal operation but they can be mapped temporarily for the storage of .TAP files (images of cassette tapes). The ZX++ includes a tape player capable of the playback of these files via DMA, and tape data has to be stored in the upper 64KB of the memory.

- An 115200 baud UART at IO addresses $7F (data) and $BF (status) for debugging. The status register includes the usual bits: Data Valid (DV), Transmitter Holding register empty (THRE), Transmission End (TEND), Framming Error (FE), and Overrun (OV):

|  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| IN ($BF) | x | x | x | OV | FE | TEND | THRE | DV |

## 3.2 Startup

During startup the boot ROM implements a master SPI controller via bitbanging and reads the Spectrum ROM from the SPI flash. After that the following code is copied to the upper RAM and executed:

```
01 FF 33    ld    bc,$33FF ; Normal mode & slow clock
ED 41       out   (c),b
C3 00 00    jp    0
```

After the `OUT` instruction the system only has the standard Spectrum ports accessible:

- ULA write register at IO address $FE and ULA read register also at $FE
- Kempston joystick at IO address $DF

Also the boot ROM is no longer present, its content being replaced by a RAM copy of the Spectrum ROM. The `JP 0` instruction transfers the control to the ROM code and a "`(c) 1982 Sinclair Research Ltd`" message finally appears at the bottom of the screen.

Well, the above booting procedure is only executed when no SD card is present, otherwise the booting involves the initialization of the filesystem data (only SD-HCs with FAT32 format are supported now) and the selection of files to load into memory. Files with .SNA or .Z80 extensions are snapshots of the Spectrum memory, usually used with emulators. Both formats can be loaded into the memory and the Z80 registers of the ZX++ board. The jumping to the loaded code is more complicated because we have to preserve the values of all registers, and also we have to set the proper interrupt mode and flip-flop flag depending on the actual snapshot.

Another supported file format is that of .TAP files. These files are digital images of cassette tapes that follows the standard format and timings of the Spectrum ROM. There are also another usual tape formats, like .TZX, more appropriate for games with turbo-loaders, but these files are quite complex and aren't yet supported. .TAP files are simply loaded into the upper 64KB of the memory (pages #2 and #3), and they remains there during execution. This memory area is read by the tape player, a small piece of hardware that generates a cassette waveform from a .TAP file when enabled (by pressing F1 in the keyboard)

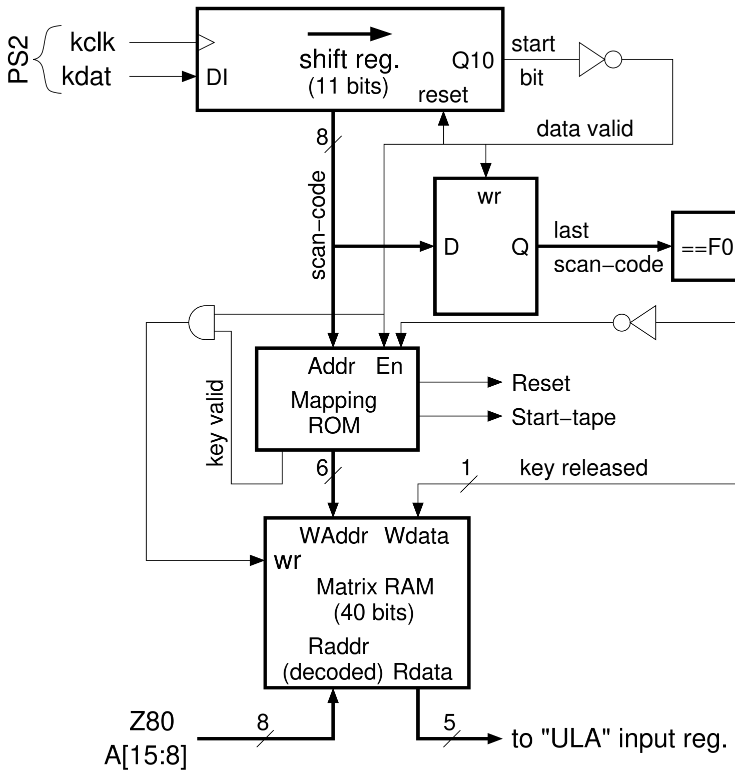


Figure 4: Block diagram of the PS2 keyboard adapter

## 3.3 Keyboard and joystick

The only remaining things we still have to implement to have a fully functional Spectrum are the keyboard and joystick logic. The keyboard is a little problematic because the PS2 interface has nothing to do with the plain switch matrix of the Spectrum, but it is still affordable. The idea here is first to receive the serial scancodes and to provide a data-valid signal for them (figure 4). Then we must take into account that released keys send the same scancode preceded by a $F0 byte, and a key-release signal have to be generated if that code is received. Finally, the switch matrix of the Spectrum is simulated by a 40-bit dual port RAM (1-bit for write, 5-bit for read) where a bit is turned off when a key is pressed or on when released. This RAM is read 5 bits at a time through the ULA's input port, the keyboard row being selected by the upper 8 bits of the Z80 address bus. Supposedly only one address bit should be low when reading, but if more than one bit is low the logical AND of the selected rows have to be returned. In this way the ZX++ gets a professional keyboard, the only problem is that the PS2 keyboard has no BASIC keywords written on the keys :( Also, two of the PS2 keys have another uses:

| Key | Effect when pressed |
|---|---|
| <esc> | system RESET |
| <F1> | Starts tape player (a .TAP file has to be loaded before) |

The joystick was a trivial peripheral, as long as it is of the passive switch type. But, as it turned out what I already had in stock was a game controller for the Gigatron computer. This controller is basically identical to a Nintendo's one, but it comes with a female DB9 instead of the weird NES connector, so it can be plugged into the FPGA board. And by an strange strike of luck the ground pin is the same as in a Kempston joystick. The FPGA has to change three pins of the joystick connector to outputs, one for providing power to the game controller chip, other for its parallel load signal, and the third for a shifting clock (the controller is basically a CD4021 chip). Inside the FPGA the clock is derived from the HSYN signal of the VGA, a load pulse is

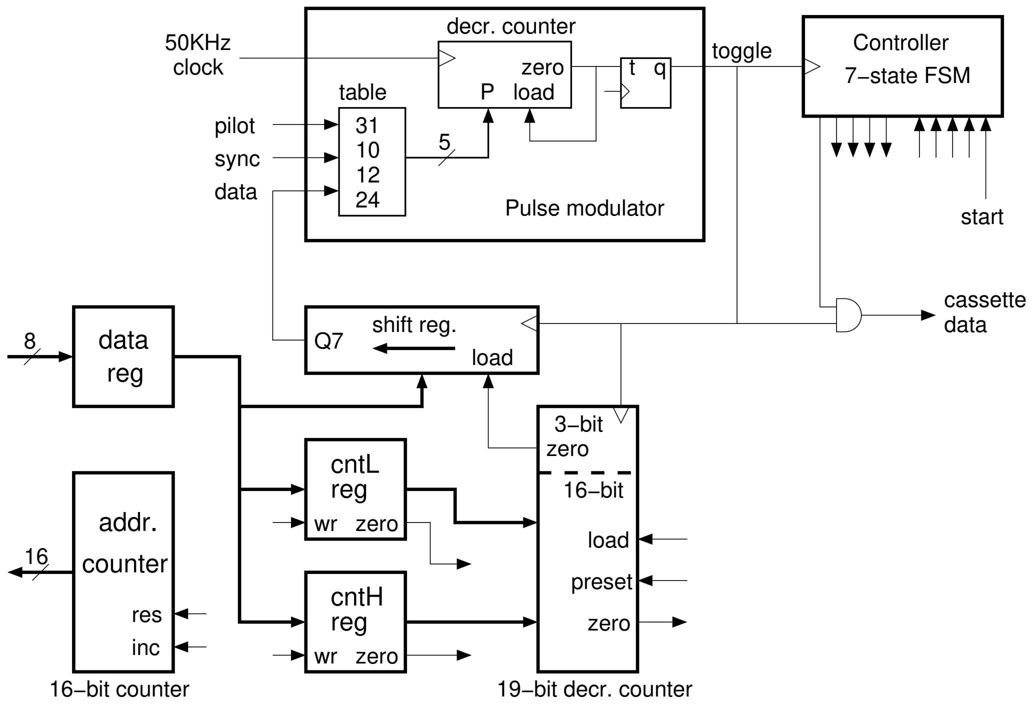


Figure 5: Block diagram of the Tape Player

generated every 8 cycles, and the incoming data stream is converted to a parallel 8-bit word that is presented as the input data of the supposed joystick port. Of course, if you only want to use a simple Atari-style joystick all this logic can be removed.

## 3.4 Playing tapes

The tape player shown in the figure 5 uses DMA to read the external memory but its data rate is orders of magnitude lower than that of the Z80 CPU, allowing for a very long latency. Therefore, data for the tape player is read when the Z80 has its MREQ signal inactive and stored in an 8-bit register where it can be read later. The CPU isn't slowed at all due to this DMA. Also, when the tape player is active the CPU clock divider is disabled and the Spectrum runs 7 times faster than the original, reducing the load time of tapes a lot. The tape player itself is basically a pulse width modulator feed from a shift register, a buffered pulse counter, a memory address counter, and a finite-state machine. All this logic takes only around 150 logic cells inside the FPGA.

The .TAP file is a collection of tape blocks, and each block beggings with a 16-bit value that holds the number of data bytes, followed by the actual data of the block. This results in a signal with the following fields:

1. A pilot tone: about 5 seconds of an square wave with $619\mu s$ low and high pulses.
2. A synchronism bit, with $190\mu s$ low and $210\mu s$ high pulses.
3. A data field where bytes are played MSB first and each bit is coded as:
   (a) Zero bit: $240\mu s$ low and $240\mu s$ high
   (b) One bit: $480\mu s$ low and $480\mu s$ high
4. An additional zero bit pulse. A terminator, probably intended for the reliable decoding of inverted waveforms.
5. An inter-block gap: about 5 seconds of silence.

These times aren't required to be very accurate because cassette tapes can also have a quite significant variation in playing speed, and BTW, the SYNC bit looks like a zero bit poorly timed by the ROM code of the Spectrum. Anyway, these times are more or less recreated from the CPU clock, first dividing its frequency by 70, and then assigning an integer number of cycles for each pulse (each cycle takes $20\mu s$ for a 3.5MHz CPU clock): 31 cycles for pilot, 10 cycles for sync (both low and high), 12 cycles for zero bits, and 24 cycles for one bits. The pilot tone can be reduced to about 2 seconds in order to save time, and the same could be done for the inter-block gaps. But this later reduction is a bit risky because some loaders expect a long gap time while doing their fancy processing.

### 3.5 Compatibility issues

As it happened to the Inves Spectrum+, the ZX++ is going to have problems with some software, especialy games. So let's point out the differences with a real Spectrum and what is expected to happen when running "too smart" programs.

- First, the Z80 runs a 2% faster than the original Spectrum 48K. This isn't supposed to be noticeable, and BTW, the Spectrum 128 also was a little faster than the 48K. But the lack of memory contention surely will make the ZX++ quite speedy.
- Next, one VGA line takes $32\mu s$ instead of $64\mu$s, so any border bitbanging code will fail for sure. You weren't supposed to draw things on the border, you nasty rascal! But the number of visible lines is also doubled, so if an application is using the border to draw just a simple horizon this can still work. Well, it depends on the time the interrupt is posted to the CPU, in this case it is posted when the VSYN signal goes low, but this can be easily tweaked.
- And also the total number of lines in a frame isn't the double. The vertical total for the VGA mode is 525 lines. This is the same as in a complete NTSC frame, not a PAL one with 625 lines. Therefore we are posting an interrupt every 16.8ms, not every 20ms. Well, we got a NTSC Spectrum instead of a PAL one.
- The floating bus isn't simulated properly, so the early Arkanoid is going to hang. Still, the good thing about FPGA designs is that they can be fixed without touching a soldering iron. But I'm not a big fan of Arkanoid... And, BTW, the fault was with Arkanoid programmers for resorting to tricks, and with the programmers of the Spectrum ROM for not providing an interrupt vector to users (I had to say it or to explode).
- Sound level differences between Speaker and cassette bits are actually simulated using PWM, with the speaker being 7 times louder than the cassette output (I like to ear the data sent to the cassette tape). No problems are expected here. But there is not feedback to the cassette input, so any code trying to determine the board issue by banging the audio bits is going to fail. This method is a crappy one because any noise at the input jack can ruin the test, so if you want to know your board issue pick an screwdriver up and break the "warranty void of broken".

## 4 The CPC464++

After the recreation of the simple ZX Spectrum it was time for another more complex machine: the Amstrad CPC-464. Let's present first what's included in that classic computer.

## 4.1 The Amstrad CPC-464

Also sold under different brand names, like Schneider or Awa, that computer included 64KB of RAM, 32KB of ROM in a single chip but on two separate memory banks of 16KB each, and a programmable sound generator: the GI's AY-3-8912. Also its video modes were more advanced that that of the Spectrum. Three basic video modes were available:

| Mode | Resolution | Text columns | Colors | Pixels per byte |
|---|---|---|---|---|
| 2 | 640x200 | 80 | 2 out of 27 | 8 |
| 1 | 320x200 | 40 | 4 out of 27 | 4 |
| 0 | 160x200 | 20 | 16 out of 27 | 2 |

The mode is selected using two bits in a Gate Array register, thus, another video mode is also possible. The unintentional mode #3 has a 160x200 resolution but with only 4 colors instead of 16, and very little interest.

Also, while the color can only be 1, 2, or 4 bits per pixel, the resolution can be changed by reprogramming the registers of the 6845 CRTC. For instance, by writing the register #1 (horizontal displayed) with 32 instead of 40 the horizontal resolution can be reduced to 512, 256, or 128 pixels.

The amount of video memory is the same for the three modes, always 16000 bytes, and its base address can be selected by writing to the 6845 registers #12 and #13 (Display Start Address). And, as it happened with the Spectrum, the mapping between video memory and screen coordinates is quite weird. The reason for this is mainly the use of a CRTC controller that was intended for displaying text in graphic modes, so, the address generated by that chip is for a "character" that is 8 lines high and 16 pixels wide (in mode #2), and another three row address bits, RA[2:0], are generated separately for the video line inside the character. These row address bits are intermixed with the character address bits in the following way:

| Video address | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A15 | A14 | A13 | A12 | A11 | A10 | A9 | A8 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |
| MA13 | MA12 | **RA2** | **RA1** | **RA0** | MA9 | MA8 | MA7 | MA6 | MA5 | MA4 | MA3 | MA2 | MA1 | MA0 | 1/0 |

Notice that A0 isn't generated by the 6845 because each character is two bytes wide. A0 is in fact the clock signal of the 6845 and it should be 0 for even bytes and 1 for odd ones. Also, the row address lines starts at A11. That means that the second video line begins 2048 bytes after the first one. Finally, we also have to remark that MA10, MA11, MA14, and MA15 lines aren't used for the address generation, so, the main idea here is to have a 16KB video region that can be mapped to addresses \$0000, \$4000, \$8000, or \$C000 depending on the MA13 and MA12 bits of the Display Start Address register.

The relationship between screen coordinates, (X,Y), and addresses is then:

$$Offset = (X/PPB) + (Y/8) * 80 + (Y\&7) * 2048$$

Where PPB (pixels per byte) is 8 for mode #2, 4 for mode #1 and 2 for mode #0, and the resulting Offset has to be added to the contents of the Display Start Address register.

The weirdness of the CPC video is also demonstrated by the way the various bits of the pixels are arranged in modes #1 and #0. In mode #1, with 2 bits per pixel, the 8 bits of a video byte are arranged in the following way:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| P0[0] | P1[0] | P2[0] | P3[0] | P0[1] | P1[1] | P2[1] | P3[1] |

Where P0-P3 are the pixels starting from the left side of the screen, each of them with 2 bits.

And for mode #0, with 4 bits per pixel:

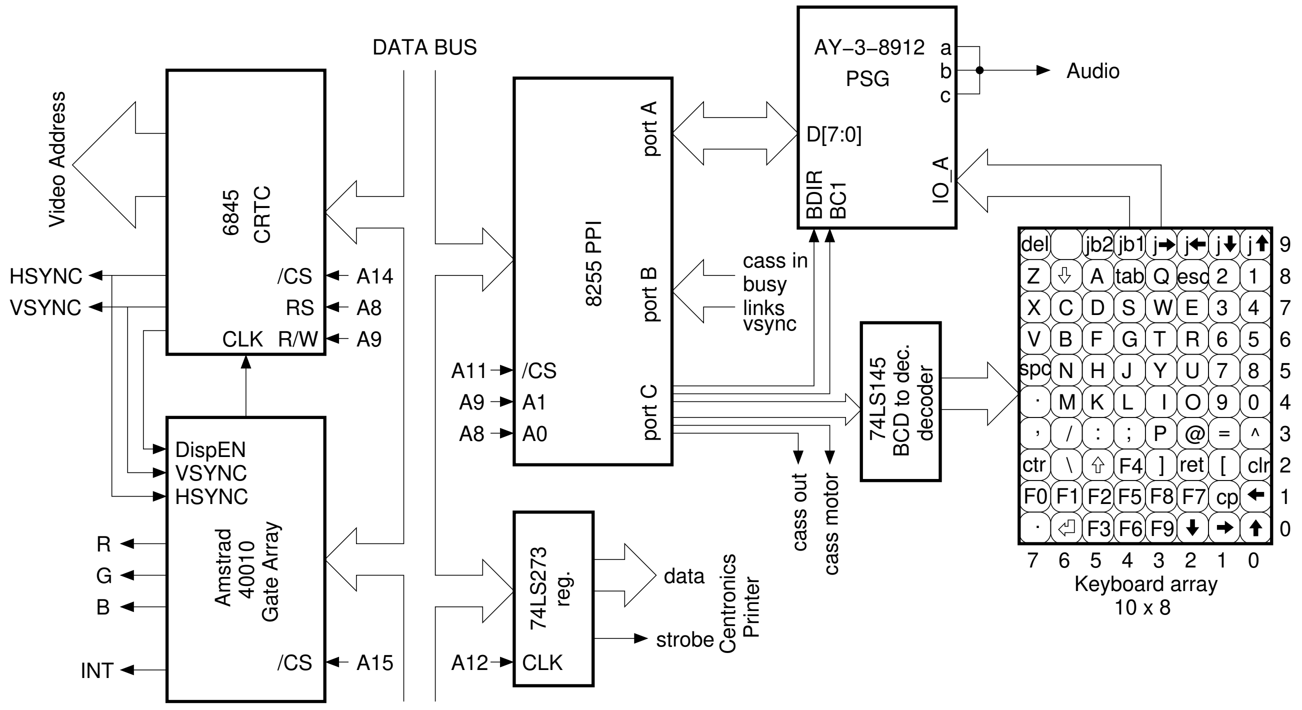


Figure 6: I/O structure of the CPC464

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| P0[0] | P1[0] | P0[2] | P1[2] | P0[1] | P1[1] | P0[3] | P1[3] |

In this case we have only 2 pixels but with 4 bits each.

In mode #2, with only 1 bit per pixel, at last the pixels are ordered starting with the leftmost pixel at the MSB bit.

Pixels can select their colors from a palette RAM inside the Gate Array. This RAM has 16 entries of 5 bits each, meaning we have up to 32 possible colors, but the actual number of colors is reduced to 27 because the way the analog R, G, an B signals are generated, and therefore 5 colors are repetitions of other colors. R, G, and B are digital outputs that can be in three possible states: high, low, or floating. When floating a resistor divider generates an intermediate voltage resulting in a 50% level of color at the monitor cable. There isn't an straight relationship between color numbers and RGB states, so, a translation ROM is also needed after the palette RAM. In video modes #1 and #2 only the first 4 or 2 entries of the palette are used respectively.

The registers in the Gate Array are write-only and 5 bit wide. They gets written when the Z80 executes an OUT instruction with the bit A15 of the address bus low. Also, the bits 7 and 6 of the data are used to select which register is to be written. The register map is then:

| D7,D6 | D5 | D4 | D3 | D2 | D1 | D0 | Comments |
|---|---|---|---|---|---|---|---|
| 00 | x | 0 | Palette Index | | | | PEN selection |
| 00 | x | 1 | xxxx | | | | BORDER selection |
| 01 | x | Color select | | | | | Color write |
| 10 | x | INT counter reset | H. ROM en | L. ROM en | Mode | | Control |
| 11 | xxx | | | RAM bank | | | Outside GA, Not used in CPC464 |

So, for instance, the way the palette entry number 1 is written with color 15 could be:

```
ld bc,$7F01 ; write index #1
out (c),c   ; register B is on address bus high (A15 low)
ld c,$4F    ; write color $F
out (c),c   ; register B is on address bus high (A15 low)
```

In addition to the selection of the video mode (bits #1 and #0), the control register of the GA also host two important bits related to ROM banking. If bit #2 (Low ROM Enable) is low any memory read in the $0000 to $3FFF range will select the lower 16KB of the ROM instead of RAM. In the same way, if bit #3 is low (High ROM Enable) any memory read in the $C000 to $FFFF range will select the high 16KB of the ROM instead of RAM. Writes are always directed to RAM.

The last function performed by the gate array is the generation of interrupts. For this purpose it includes a 6-bit counter driven by the horizontal sync. signal. When the counter reach the value #52 it gets reset and an interrupt is requested, resulting in about 300 interrupts each second. This counter is also reset by the vertical sync signal, or when the control register of the GA is written with the bit D4 high (this bit is not stored). Well, the vertical sync is actually delayed $128\mu s$ (two video lines) before requesting its interrupt and this delay seems to be needed to allow the detection of the vertical sync signal by polling.

The 6845 CRTC has 18 registers, most of them write-only. In order to write a register its corresponding index has to be written first by means of an OUT instruction with A14, A9, and A8 low (address $BCxx). Then A8 changes to high on a second OUT instruction with the data to be written into the previously selected register. For instance, the following code writes the CRTC register #1 with the value 32:

```
ld  bc,$BC01  ; CRTC index write
out (c),c     ; index 1
ld  bc,$BD20  ; CRTC data write
out (c),c     ; value 32
```

Some registers of the CRTC can be read by IN instructions with A9 high, but this is rarely done in practice, so, the CRTC is more or less a write-only peripheral. BTW, any OUT instruction with A14 low and A9 high (at $BExx or $BFxx, for instance) will result in electrical contentions in the Z80 data bus because Amstrad designers completely neglected to include the /RD and /WR signals in the decoding logic of the CRTC. And yes, shortcircuits are going to last less than one microsecond and nothing is going to catch fire, but they are a nasty thing to have in the circuit anyway. The fact that these contentions could have been avoided without any extra hardware (by just connecting the RW pin of the CRTC to the /WR signal of the Z80 instead of A9) makes me think the circuit designer was a total inept.

The Diagram of the I/O structure of the CPC464 is shown in figure 6, were the already mentioned Gate Array and CRTC are in charge of video generation. Apart from a dumb write-only register for a parallel printer interface, everything else is managed from an 8255 PPI, and this includes a bitbanging interface to an AY-3-8912 Programmable Sound Generator, that, in addition to the audio output, is also used for the reading of the keyboard. The direction of the Port A of the PPI, has to be established according to the reading or writing of the PSG registers, but that's the only port whose direction isn't fixed. Port B is always used as input and port C is always output, providing a total of 8 input and 8 output signals which are:

| Port B input | Signal | Comments |
|---|---|---|
| PB0 | VSYNC | Vertical Synchronism from CRTC |
| PB[3:1] | Manufacturer Links | 3 PCB jumpers for manufacturer code |
| PB4 | 50Hz / 60Hz | PCB jumper for PAL/NTSC (0=NTSC) |
| PB5 | /EXP | Expansion connector signal |
| PB6 | Busy | From the Printer interface |
| PB7 | Cassete Data Input | |

| Port C output | Signal | Comments |
|---|---|---|
| PC[3:0] | Keyboard row select | Only values 0 to 9 valid |
| PC4 | Cassette Motor | Motor On when 1 |
| PC5 | Cassete Data Output | |
| PC6 | BC1 | Control of the AY-3-8912 bus |
| PC7 | BDIR | |

The input signals include 4 PCB "links" that are jumpers to ground. A link soldered sets the corresponding input as 0. One of these links is used to select the monitor frequency, while the other three have the effect of changing the manufacturer name in the startup text. A value 111 selects "Amstrad", while other combinations are reserved for other manufacturers like "Schneider".

In the port C, the two higher bits are used to control the bus of the AY-3-8912:

| PC[7:6] | Operation |
|---|---|
| 00 | NOP |
| 01 | Read PSG register |
| 10 | Write PSG register |
| 11 | Write PSG index |

Again, the registers in the PSG have to be selected prior to any read or write. The direction of the PPI port A has to be programmed as input for PSG reads or output for writes (index or register).

The last thing worth a mention is the way the joystick is managed. Its 6 switches are equivalent to 6 keys in the last row of the keyboard matrix. Interestingly, a second joystick can also be chained to the first one, but for this case its switches are connected in parallel with those of the keys "5", "6", "R", "T", "G", and "F", and therefore generate the same codes as the mentioned keys.

## 4.2 Design of the Amstrad CPC-464 replica

### 4.2.1 Video generator

In the CPC case the amount of video memory is a bit too much to be placed inside the FPGA, because 16KB is all the available RAM, and we still need some for a boot memory, so, a "ghost CPC-video" is not possible. We have to use the external RAM as the video memory and we must arbitrate the sharing of the memory between the CPU and the video controller.

As usual, I wanted a VGA compatible video signal. The VGA mode, with a 640x480 resolution, is enough for the CPC recreation, yet, with this mode no extra horizontal pixels remains for the drawing of a border, so, the border would appear in the top and bottom sides of the image, but not on the left and right sides. In order to have the same border than the PAL video signal of the CPC a 32MHz clock should be used for the VGA, but the use of a non standard clock will surely result in sampling artifacts on LCD displays, so, I think it's better to fill the entire screen with a good quality image and to loose the lateral borders than to keep the borders but at the price of some vertical ghost lines. Thus, at the end I settled for a standard 25MHz clock rate.

The VGA video runs twice as fast as the PAL signal of the CPC and it isn't interlaced. In order to reproduce the 200 lines of the CPC each VGA line has to be displayed twice, and still we have 80 remaining lines to be assigned to the vertical borders.

The video controller must reproduce the main features of the Gate Array and the CRTC, at least to some extent. On the other hand it also makes little sense to try to recreate all the programmability of the CRTC when all the timings of the VGA video are fixed. In fact only 4 of the 18 registers of the CRTC had to be recreated. These are:

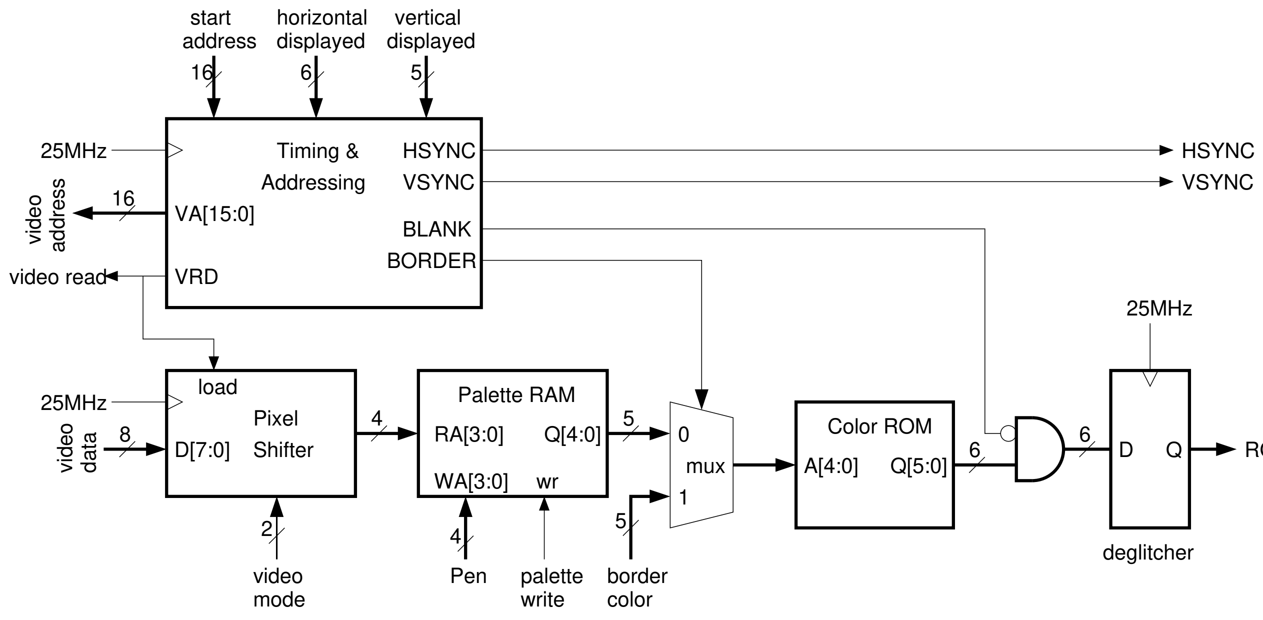


Figure 7: Diagram of the video generator in the CPC464 recreation

- Register #1: Horizontal displayed. The value of this register, multiplied by 2, defines the horizontal resolution of the screen in bytes. 40 is its maximum value (640 pixels in mode #2), but some games ("La Abadía del Crimen" or "Starwars" for instance) reduce the resolution to 512 pixels (or 256 pixels in mode #1) by writing this register with 32. When this is done the timings of the VGA signal are adjusted automatically to generate some lateral borders with the unused pixels of the lines. No other timing registers have to be written (these registers aren't implemented anyway)

- Register #6: Vertical displayed. The value of this register defines the vertical resolution, in text characters, of the screen. Its nominal value is 25 but some games, like those mentioned before, can reduce the vertical resolution to 24. The vertical resolution is the value of this register multiplied by 8.

- Registers #12 and #13: Display Start Address. These two registers define the beginning of the image in the memory and the CPC ROM makes use of them to implement an smart scrolling, so their recreation is mandatory.

No other CRTC register is necessary. Also, the current CPC recreation has all registers as write-only, and everything seems to be working fine (registers #12 and #13 can be read-back, registers #1 and #6 are truly write-only).

With respect to the Gate Array, we have to recreate:

- The reading of video memory every 8 clock cycles during the visible part of the line.

- The shifting of video data depending on the video mode selected (1, 2, or 4 bits per pixel).

- A 16x5 palette RAM.

- A 32x6 color ROM. The video output has 2 bits per each component, R, G, and B. But with only 3 possible values: 00, 01, and 11, which correspond to 0%, 50%, and 100% of the color intensity.

Also, the GA recreation must include a 5-bit border register, a 4-bit palette index register, a 4-bit control register, and a 7-bit interrupt counter (one bit more than the GA because 3.3ms takes now 104 display lines instead of 52).

With these considerations the video generator of the CPC recreation follows the diagram of figure 7. In this diagram the control signals are supposed to come from registers that emulate those of the CRTC and GA.

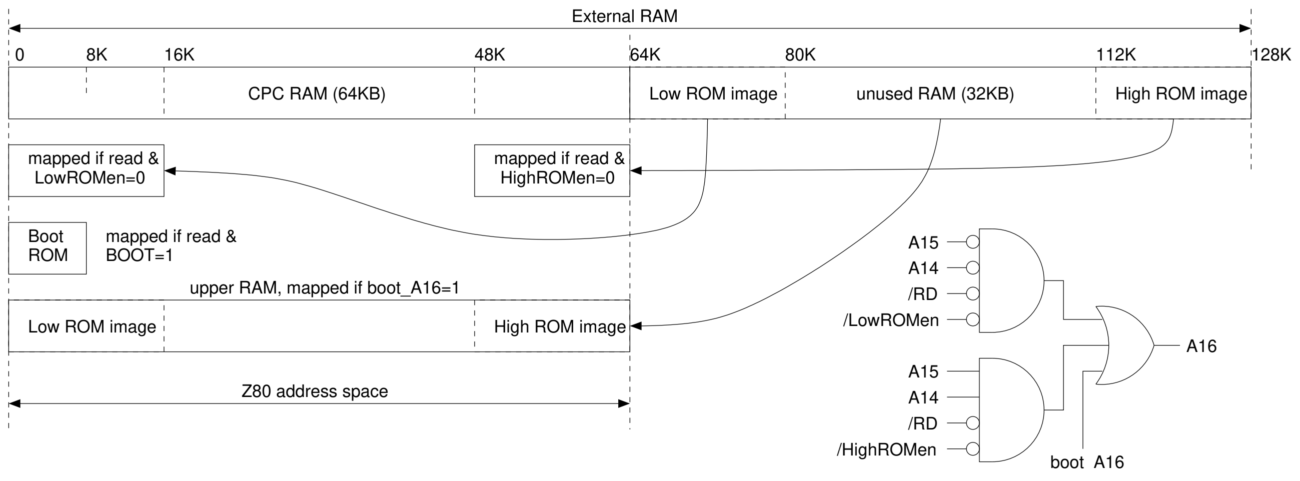


Figure 8: Memory mapping in the CPC recreation and equivalent logic.

Using fixed values for these signals instead of registers the video controller was able to display the contents of the external RAM in a VGA monitor even before including the CPU and the rest of the peripherals in the recreation. This controller used about 130 logic cells of the FPGA while, in comparison, the Z80 takes around 2200 logic cells. This looks like very little logic, indeed, and BTW, the controller includes the functions of both the CRTC and the GA, so, my early idea about the underusage of the GA seems to get confirmed.

As a last comment about video generation, the output register for the RBG signals filters out any combinational glitches the pixels could have in order to get a perfect image in the screen.

### 4.2.2 Clocking

The video generator requires a 25MHz clock and my idea here was to get all other clocks from this signal. These are a 4MHz clock for the Z80, and a 1MHz clock for the AY-3-8912.

The PSG clock only requires a simple divider by 25, but the Z80 clock is a bit more problematic. First, 4 isn't an integer divider of 25, and second, memory access have to be arbitrated between the CPU and the video controller. The first problem was solved by the same divide by 25 counter, where instead of a single pulse we generate 4 pulses on counts #0, #6, #12, and #18. This clock signal isn't strictly periodic, but that's no problem. Also, if one of these pulses happens at the same time than a video read it gets delayed an additional clock cycle, thus, avoiding memory contention. The resulting Z80 clock is aperiodic, but on average it has 4 pulses every microsecond, the same as with a perfect 4MHz clock.

Also a fast clock can be selected during boot time. In this case All clock cycles but video reads are translated into clock pulses for the Z80. This is equivalent to having a 23.1MHz clock in the CPU or to run 5.77 times faster than normal.

I must remark this is possible thanks to the fast memory of the SIMRETRO board that has not problem when running at 25MHz, and also thanks to the timing of the TV80 core where the MREQ and IORQ pulses last only a single clock cycle (but they are spaced apart to mimic the cycle count of a real Z80)

### 4.2.3 Memory banking and boot

The SIMRETRO board includes a 128KB external SRAM in addition to the internal 16KB of the FPGA. My idea here was to use the external SRAM for both the 64KB of RAM and the 32KB of ROM of the CPC-464. The ROM contents have to be preloaded into RAM before starting the CPC, and in order to do that, a small boot ROM is also included, this time into the internal memory of the FPGA because that memory can have an initial content defined in the FPGA configuration bitstream. In summary, the intended mapping and the related logic is that shown in figure 8. The ROM images are loaded into the upper half of the RAM by means of the boot code and then, when reads are performed into the enabled ROM areas the bit A16 is set high, resulting in reads of the upper memory. Writes, on the contrary, are performed into the lower half of the RAM unless A16 is forced high by a bit in a control register. This is precisely what the boot code does when it reads the ROM images from an SPI flash and stores them into the upper RAM.

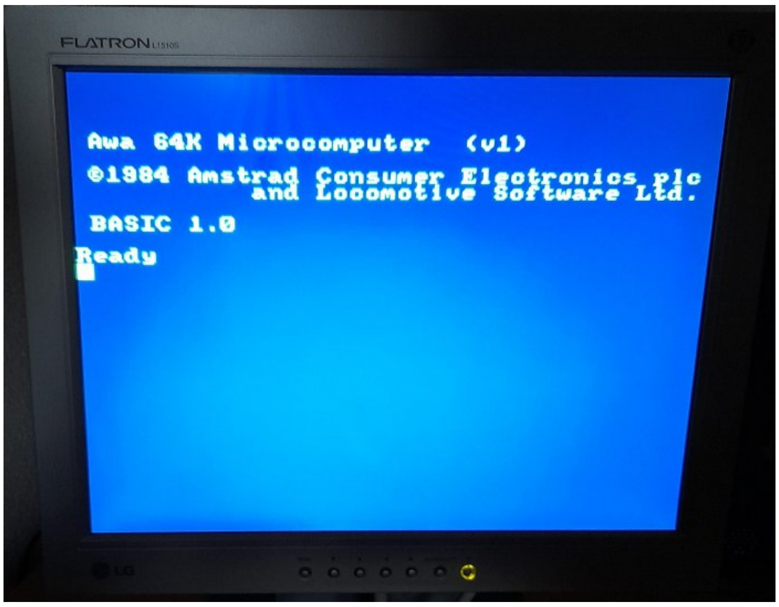
So, we need some storage for at least two bits: BOOT, that enables the boot ROM, and boot_A16, that allows to write the upper RAM. The idea here is the same as with the ZX++: to include a boot I/O register that is only accessible during boot and then disabled. The I/O address I chose for it was the same as that of the Centronics printer, mainly because no printer is going to be recreated here. Anyway, the decoding of the boot I/O register includes the BOOT bit, so the printer interface could still be recreated if desired. The bits of the register are:

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| OUT, IN ($EFxx) | MOSI | SCK | $\overline{\text{CS1}}$ | $\overline{\text{CS2}}$ | BOOT | A16 | FAST | MISO |
| | R/W | R/W | R/W | R/W | R/W | R/W | R/W | RO |

Here, in addition to the already mentioned BOOT and A16 bits, we got the signals of the SPI bus and also another control bit, FAST, which selects a fast Z80 clock when one. After a reset all R/W bits are loaded with 1, meaning the boot ROM is active, the upper half of the RAM is selected, and the CPU runs with a fast clock. Thus, the boot program can proceed to read the images of the Amstrad ROMs from the SPI flash, then to select the lower RAM, to place a trampoline code into that RAM, and to jump to it. The trampoline code is listed next:

```
ld     bc,$EF30     ; SCK,MOSI: low, CSs high, no boot, slow clock
out    (c),c
rst    0
```

After the OUT instruction the boot register can't be written again, and the RST 0 instruction transfers the execution to the Amstrad ROM. An snapshot of the first working VGA screen is shown next:



```
Awa 64K Microoomputer  (v1)
©1984 Amstrad Consumer Eleotronics plc
          and Looomotive Software Ltd.
BASIC 1.0
Ready
```

Well, the system at that point included only the video controller, the CPU, the memory, and the boot ROM and I/O. The rest of the peripherals were still missing. Interestingly, the manufacturer is listed as Awa, not Amstrad. Also the text color was cyan, not yellow as it should be. There was an error in the bit order of the pixels in video mode #1 that was easily detected and corrected.

### 4.2.4 The 8255 PPI and keyboard

In order to interact with the Locomotive BASIC a keyboard interface is mandatory, but looking at the block diagram of figure 6, we can see that before the keyboard the 8255 PPI and the AY-3-8912 PSG have to be recreated. Well, there is a shortcut approach: to bypass the PSG and to connect the keyboard row directly to the port A of the PPI. In this way the PSG can be left as the final touch of the CPC recreation.

The PPI doesn't require a detailed recreation because all its ports are used as simple I/O (without strobes or interrupts). Port C (I/O address $F6xx) is always output and requires an 8-bit register that can also be addressed as 8 single bit outputs if the control register (I/O address $F7xx) is written with any data with D7 as zero:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | x | x | x | Bit | | | Data |

Port B is always input and its various signals only have to be routed to the CPU input data bus when a read is performed at I/O address $F5xx.

Port A is the interface to the PSG data bus and sometimes it is an input and others an output. But, as any FPGA block must have separate inputs and outputs (no bidirectional buses are possible), two ports have to be implemented: one for reading and another for writing. These two ports (at I/O address $F4xx) will be connected to the two data ports of the PSG, but meanwhile the input port A will be routed to the keyboard.

The keyboard interface is basically identical to that of the ZX++ (figure 4), the only differences being the amount of RAM, this time up to 80 bits, the mapping of keys, and the reading of the matrix RAM: now the row is selected using the 4-bit combination of bits in the lower port C of the PPI. The mapping of keys was a little tedious, but at the end we were able to type basic programs on the CPC recreation and to execute them. Also, the startup message now listed the computer as Amstrad (or Schneider or some other manufacturer, depending on port B inputs). Only the sound was still missing.

### 4.2.5 The AY-3-8912 PSG

For the PSG block I resorted to the JT49 implementation of José Tejada. That Verilog module was in fact for a compatible Yamaha YM2149 PSG that has only minor differences with the AY-3-8912 and potentially a better sound quality (wave envelopes have a 32-level volume control instead of 16-level, everything else is the same as in the AY-3-891x). Of course, that module provided an 8-bit input port for the keyboard, in addition to a 10-bit digital sound output.

I was planning to use PWM modulation for audio and 10 bits of resolution would result in a somehow low carrier frequency (24.4kHz). Instead of truncating the audio bits I recalculated the exponential table of the module for a maximum level of 85 instead of 255, so, when the three channels are added together the maximum value will be 255 and an 8-bit PWM will be enough. The resulting carrier is going to be 98KHz, high enough to avoid audible aliasing or PWM sidebands. And, anyway, 8-bit is more resolution than anything these primitive sound generator users ever dreamed.

I tested the resulting sound using the BASIC "SOUND" command and it was correct, but, in order to better appreciate the PSG capabilities I wrote a .PSG file player (see annex 7.2) and checked it with a fragment of one of these files (.PSG files are a simple list of register values separated by delays). Well, notes were grossly detuned because the file was for a ZX Spectrum with 1.75MHz clock instead of 1MHz, and the tempo was a little fast due to the 60Hz of the VSYNC signal instead of 50Hz, but the sound was clearly the expected (an AY-3-8910 emulator for PC also played the file with the same results after adjusting frequencies)

### 4.3 CPC-464++ conclusions

The CPC recreation was easily accomplished, mainly thanks to the previous ZX++ experience, and results were rewarding. The fact you can type BASIC programs just like any other text instead of looking for keywords through the keys is really very convenient.

The weak side of this computer is again tape storage. With cassette signals routed to FPGA pins programs can be loaded, and potentially stored, using external devices like TZXduino. But to use external hardware while an SD card connector is already present in the board looks non-optimal. I though about a similar approach as the integrated tape player for the ZX++, but I faced some problems: there isn't enough free memory to store a complete tape image, to read an SD card with a FAT filesystem is something too complicated for a simple state machine controller and it would also require buffering to cope with latency, and the last one: tapes are very slow, so, something is needed in order to speed up their loading.

So, maybe, the next improvement would be the recreation of a floppy controller in a CPC664. (Not really for a hurry)

## 5 Cassette tapes again

The tape player of the ZX++ has an important limitation: it only allows to play .TAP files up to 64KB in size. But .TAP files can't be used with other computers than the ZX Spectrum nor they support nonstandard tape formats. On the other hand there is another tape image format, the .TZX file, which covers all these arbitrary tape encodings and formats. The downside of these files is their complexity. In order to cover all possible cases a lot of different blocks with all their metadata have to be interpreted in the tape player.

And also there should be an storage device present to keep these files on it. The ideal choice would be an SD card, yet, these cards are a pain in the neck for programmers unless you get all the code already written in an Arduino library.

And this is more or less what the programmers of the TZXduino did. The TZXduino is a system build with Arduino blocks that plays .TZX files (and also .TAP files) from an SD card. The designers used an Arduino-nano module, an SD card shield, and an I2C graphic screen. The only parts they put on their own were the buttons and the audio amplifier, and BTW, they got it completely wrong. The LM386 wasn't needed at all because you already have a wave with 5V amplitude. To amplify such a wave by a factor of 200 is simply nonsense, the amplifier will go into clipping for sure. Also, 5 volts is a too low voltage for the amplifier supply, resulting in an "amplified" wave with less amplitude than the input, and if you need a low output impedance to drive an speaker a pair of transistors would be enough. The schematic only shows the total incompetence of its designers about analog electronics. Yet, it works.

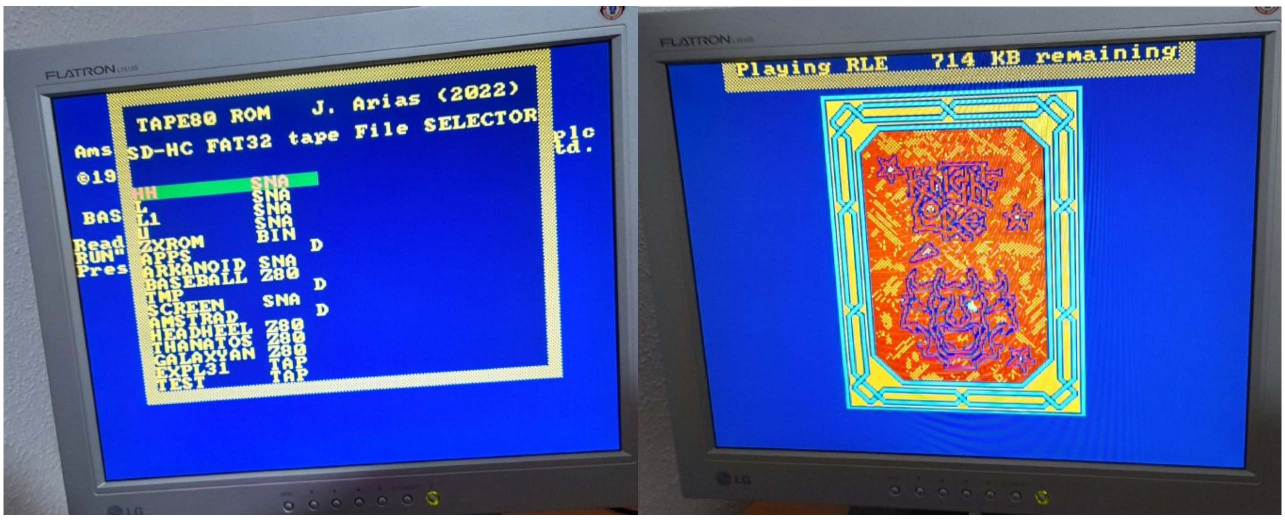
My idea here was to include a TZXduino equivalent into the free space of the FPGA. First I though about using my own CPU design, but in order to reuse some assembly code I ended choosing a Z80 as the CPU of the tape player. The FPGA board already has an SD card connector, so, only the screen and the buttons of the TZXduino are missing. But wait, There is an VGA video output and a PS2 keyboard input, and that's a lot more than the TZXduino has for user interface. The only problem here is that this interface has to be shared with the computer replica (ZX or CPC) that is also integrated in the FPGA.

The memory available for the tape player is also thigh, so, the player program should be written in assembly language in order to reduce its size (I tried the sdcc compiler, but the code it generated was larger than the whole memory of the FPGA) I needed about 3KB for the code of the player and this includes all the required FAT32 handling and an interactive file selector. The player must also include a video generator and this is going to require also a lot of memory unless it allows only a text mode. In this case a 32x24 screen only requires 768 bytes of video RAM, but another memory is needed for the character table. The coding of the ASCII characters

32 to 127 as 8x8 pixels also requires another 768 bytes. I decided to use the same memory for video, character generator, and program storage, and all this memory is only 8KB, leaving free half of the FPGA RAM for its use in the replica computer.

The video controller of the tape player is an slave one. That means that instead of generating the horizontal and vertical sync pulses these signals are inputs and the internal horizontal and vertical counters gets preset by them. Also, in the tape player the MSB of the character code is used as a remark bit that can change the color of the selected text. Not only that, characters with ASCII codes below 32 are transparent. This means that the video signal of the replica is passed to the monitor instead of the text of the tape player. In this way the size and place of the player window can be selected as desired, we only have to write all the other characters with zeros or any other value under 32.

The tape player also has a simple keyboard interface for the receiving of key scancodes. This interface is connected in parallel with the keyboard of the replica computer and this could cause interference when typing to the tape player. In order to avoid this the player is able to "mute" the keyboard of the main computer when it pops on the screen. The <sys_req> key, that is unused by the replica, has the effect to show and hide the screen of the player computer. This screen also pops up when the signal of the cassette motor is turned on in the Amstrad replica. The following snapshots give an idea of the interface. Notice that the 32x24 character window of the tape player begins in the border area of the CPC image and therefore its first two lines don't hide any text or graphics the main computer is displaying. The selected file name has all its characters, spaces included, with the MSB bit in one.



```
TAPE80 ROM   J. Arias (2022)
SD-HC FAT32 tape File SELECTOR

              SNA
HH            SNA
L             SNA
L1            SNA
U             BIN
ZXROM              D
APPS
ARKANOID      SNA
BASEBALL      Z80
                   D
TMP
SCREEN        SNA
AMSTRAD            D
HEADHEEL      Z80
THANATOS      Z80
GALAXYAN      Z80
EXPL31        TAP
TEST          TAP
```

```
Playing RLE    714 KB remaining
```

These images shows the Amstrad replica loading a tape image from an SD card (with a wrong color palette, BTW. There were still bugs around the interrupt generation logic in the Amstrad replica). But the format of the file wasn't a TZX. The code for the playing of these files was still awaiting development. The tape file format was in fact a much simpler one where each byte contains the logic level of the cassette audio along with a 7 bit counter that indicates how many samples this level is going to last:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| Audio level | Number of samples | | | | | | |

The sampling rate is 44100 S/s, or 22.67$\mu s$ per sample, so this can be though as an slightly compressed audio file format. I call this data format RLE (Run Length Encoded), and I developed it as an intermediate representation of the cassette data in between the .TZX file and the actual playing hardware. In fact, these bytes are precisely what the timer of the player is expecting to get at its input register. RLE files are longer than .TZX, but this doesn't matter much for a multi gigabyte SD card. A single audio cycle takes two bytes in the RLE file while in the .TZX it was represented by just one bit. Also the pilot tones of the tape blocks and gaps add more

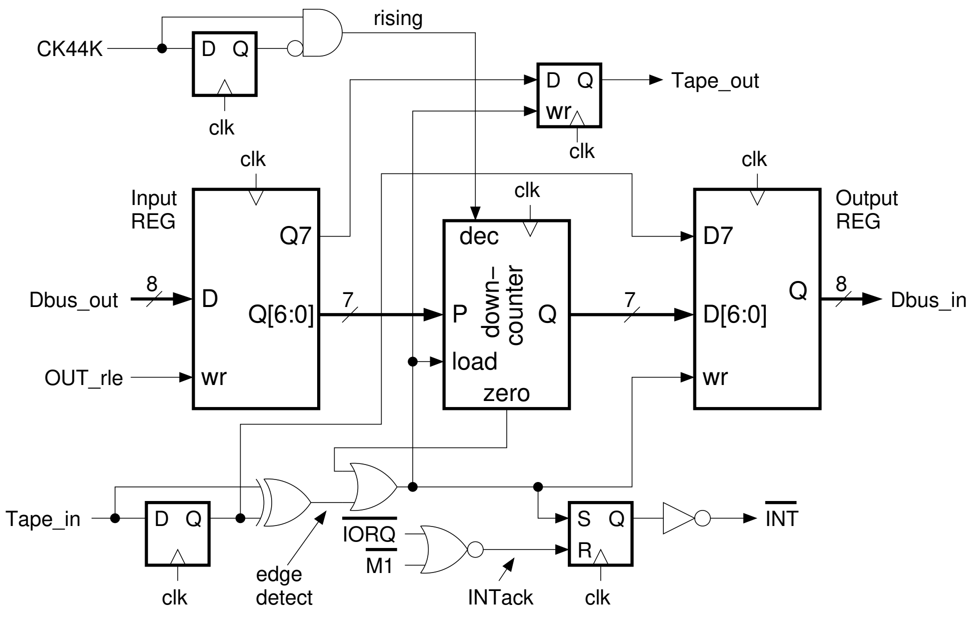


Figure 9: Diagram of the RLE timer peripheral of the tape player

bytes to the RLE file while these take almost no data in the TZX case. In overall, the RLE file is about 20 times the size of the TZX one, but in turn it is also about 1/16 of a .WAV audio file.

The 44.1kHz clock is an input to the tape player and in the CPC recreation it is derived from the 4MHz clock of the CPU by dividing it by 91. The interesting aspect of this configuration is the fact that the player is counting cycles of the replica clock instead of absolute time. So, if the speed of the main CPU clock is changed, the the frequency of the player audio is also changed accordingly and the data is still recovered reliably. Therefore, in the CPC recreation I resorted to speed up the CPU clock when the cassette motor is active. This multiplies the effective clock frequency by about 5.8 and the time required for the loading of a cassette image is divided by the same factor, which is a very desirable feature for the annoyingly slow tapes.

Another function the player is able to perform but still needs more development is the recording on tape files form the audio data of the replica computer. The basic idea here is to measure the time between changes in the the cassette signal. The captured values have to be collected in a buffer and transferred to an RLE file in the SD card. This will allow the saving of BASIC programs.

The other notable peripheral included in the player is a simple but fast SPI controller. In this device transfers are always 8 bit long and the polarity an phase of the clock are fixed. The frequency of the SPI clock can be selected to be the 25MHz of the main clock or 1/256 of this value. The SD card is configured for its use in the SPI mode using the slow clock, and then the frequency is changed to fast. With the fast clock the card has a peak bandwidth of 3.125 MByte/s, much more than really needed for the emulation of a less than 1.8 kilobit per second tape, even when it is played about 6 times faster than normal.

A more detailed description of some of the tape player blocks follows.

## 5.1 The RLE timer

The block diagram of the RLE timer is shown in figure 9. Here all the audio playing and recording is handled via interrupts. When playing, the *Tape_in* input remains constant and no pulses are generated at its edge detector output. Therefore, the 7-bit timer downcounts until it reaches zero and then it gets reloaded from the input register at the same time it requests an interrupt to the Z80. The interrupt routine then gets another byte from a memory buffer and writes it to the input register while the main program keeps reading the SD card and filling the buffer. An 1KB buffer is enough to store the samples for the equivalent of 512 bits of data or an average of 0.28 seconds of audio (48ms of audio with a fast clock). This buffer allows for a continuous playing in spite

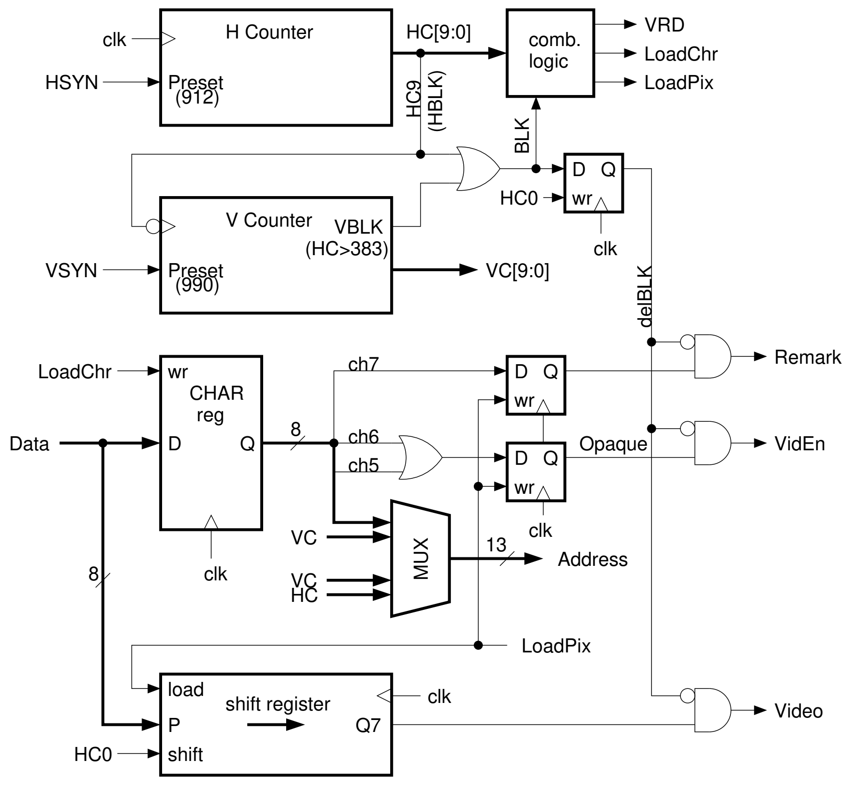


Figure 10: Block diagram of the slave video generator for the tape player

of the SD card latency (from the read sector command to the actual data there is a variable delay ranging the milliseconds).

For tape audio recording the input register has to be loaded with its maximum value (127). Then, each edge in the tape input signal stores the current counter value in the output register along with the tape level and reloads the counter. And, of course, it also requests an interrupt. In the interrupt routine the output register is read, its value subtracted from 127, and the result stored in a memory buffer while the main program keeps writing the buffer to a file in the SD card.

## 5.2 Text mode video

The video controller of the tape player is a little unusual. First, it is intended to sync to another video signal, and second, it is designed to display text with some additional features instead of graphics. Its block diagram is shown in figure 10, where we should notice the *CHAR* register. This is where the ASCII codes of the text characters are stored. Other text mode video controllers include a character generator ROM with the pixels of the characters, but in this case the character generator data is stored in the same RAM memory as everything, and therefore, the memory has to be read twice for each 8-bit segment of a character: first the ASCII code is read from the video region of the memory and stored in the temporary *CHAR* register. Then, the memory is read again using the previously loaded ASCII code as part of the address and a 8-pixel data is finally transferred to the shift register.

In order to illustrate the inner workings of the video generator the chronograph of figure 11 shows the beginning of a video line. Here the horizontal counter, *HC*, was previously loaded with the value 912 when the HSYN pulse was active. Then it spent some time incrementing, and when it rolled over to zero the blank signal, *BLK*, is deasserted and the visible part of the line begins. But before shifting out any data we have to retrieve it from memory. The video read signal is thus asserted for the duration of two clock cycles. During this time the clock of the Z80 is stopped and the address presented to the memory comes from the video generator. During the first cycle the address is for video memory and the destination register for the data is the *CHAR* register.

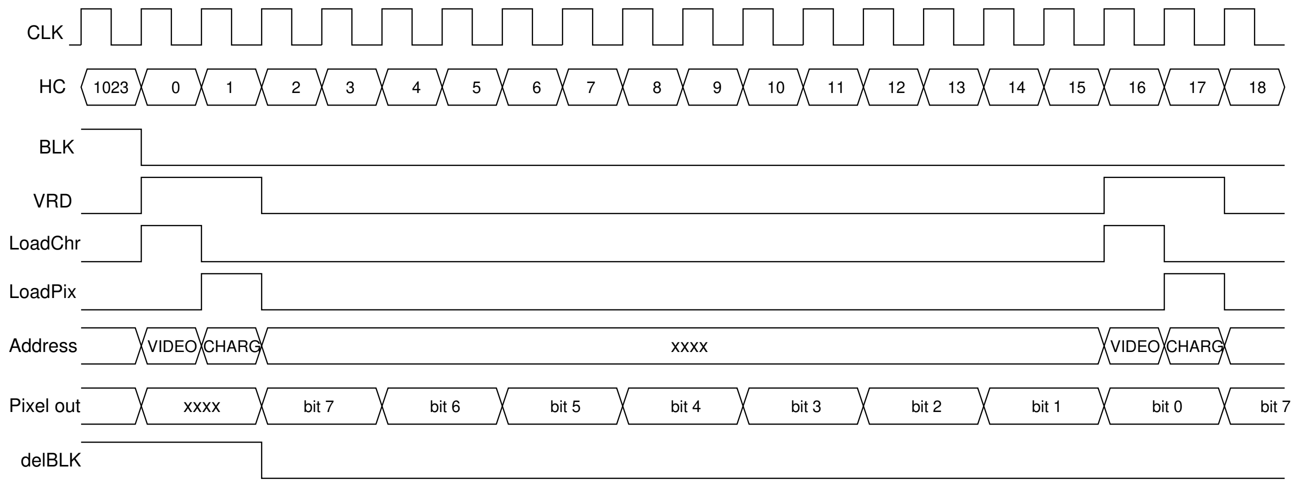


Figure 11: Chronograph of the video signals in the tape player

The video address is:

| A12 | A11 | A10 | A9 | A8 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | VC[8:4] | | | | | HC[8:4] | | | | |

Where *VC* and *HC* are bits of the vertical and horizontal counters. Then, in the next cycle the address presented to memory is for the character generator table, and it is:

| A12 | A11 | A10 | A9 | A8 | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | ASCII code (7 bit) | | | | | | | VC[3:1] | | |

The data read during this second cycle is stored in the shift register of the video generator, and pixels can start to come out towards the screen.

Notice that VC[0] isn't used in the generation of memory addresses and that means that video lines are displayed twice. HC[0] is also used as an enable signal for the shift register, so, 8 pixels take 16 cycles to be shifted out. The net result is to have each pixel of the tape player as a 2x2 VGA pixel block. The 3 MSB bits of these addresses are 111 for the video region and 000 for the character generator. Therefore, the character generator is located at the beginning of the memory, but when the character code is lower than 32 the *VidEn* output is low, disabling the tape player video. So, the first 256 bytes of the memory are never displayed on the screen and they can be used for other purposes, like for the starting code of the Z80 or its interrupt routine.

Also, the video memory starts at the last kilobyte of the memory but only 768 bytes are displayed, leaving 256 unused bytes at the end that can be assigned for instance as the stack area.

So, at the end we got more or less the following memory map:

| Start Address | Length | Use |
|---|---|---|
| $0000 | 256 | Startup code, Interrupt routine |
| $0100 | 768 | Character generator table |
| $0400 | Variable (~2280) | Application code |
| ~$0CE8 | Variable (~1580) | Application variables |
| ~$1314 | Variable (~1260) | Free |
| $1800 | 1024 | Audio buffer |
| $1C00 | 768 | Video memory |
| $1F00 | 256 | Stack |

Of course the memory occupation will depend on the actual code and the free space is going to be reduced when more functions are added to the device. It's worth to mention the big variable area, most of it dedicated to store a sector of the FAT table (512 bytes) and directory entries (1024 bytes)

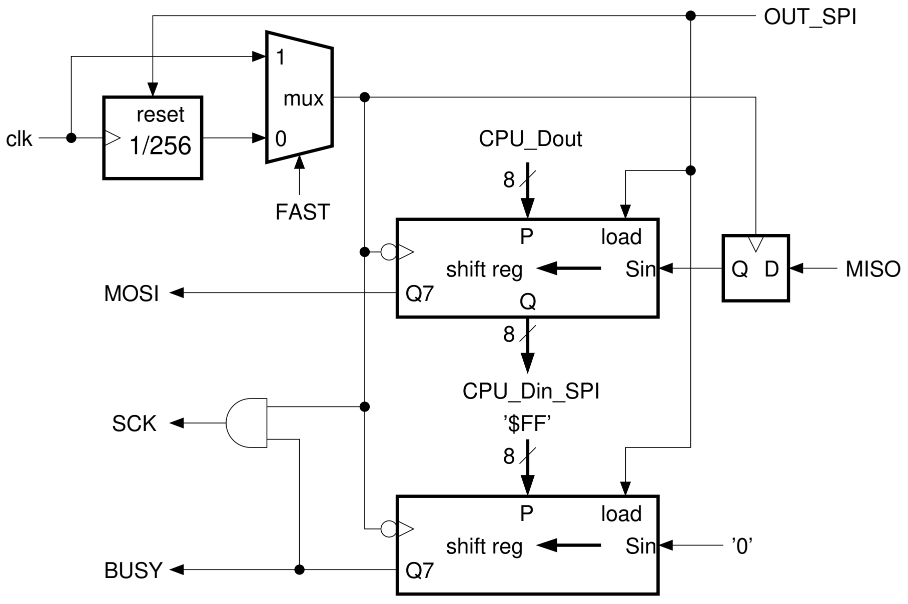


Figure 12: Diagram of the SPI peripheral

## 5.3 The SPI controller

The block diagram of the SPI peripheral is shown in figure 12. It is a simple but very effective interface. Only SPI mode #0 is supported and that means output data have to be shifted on the falling edges of the clock but input data have to be sampled on rising edges. Therefore an additional flip-flop is included to sample the *MISO* signal. The clock can be selected from the main clock (25MHz) or from an 1/256 prescaler. The slow clock is going to be used during the initialization phase of the SD card, while the fast clock is intended for the rest of the SD card operation. The controller includes two 8-bit shift registers, one is for the actual data while the other is used for control. This last register gets loaded with all ones when an OUT (SPI) instruction is executed, and consequently, sets the *BUSY* signal during 8 *SCK* cycles. The status of the *BUSY* signal can be read from an input register, but when the clock is fast there is no need for any polling because an SPI transfer takes less time (8 cycles) than the fastest polling period, so, an SPI transfer could be performed with just two assembler instructions:

```
OUT (SPI),A    ; Send data in Acc
IN  A,(SPI)    ; Received data in Acc
```

Of course, with the slow clock the polling is absolutely necessary because now the SPI transfer takes 2048 cycles.

## 5.4 I/O map and summary

In the tape player there are only 4 I/O addresses used:

| Addr[1:0] | type | register |
|---|---|---|
| 0 | R/W | CONTROL/STATUS |
| 1 | R/W | SPI |
| 2 | RO | KEYBOARD |
| 3 | R/W | RLE TIMER |

Only the *CONTROL/STATUS* register needs further description. Its bits are:

|  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| OUT | GRAB | FAST | /CS1 | /CS2 | x | x | x | x |
| IN | GRAB | FAST | /CS1 | /CS2 | 0 | MOTOR | BUSY | KVAL |

Where *GRAB* is an output bit that mutes the keyboard in the replica computer, *FAST* and *BUSY* are used in the SPI controller, */CS1* is the chip select signal of the Flash SPI memory of the board, */CS2* is the chip select signal of the SD card, *MOTOR* is the signal of the replica computer that goes high when it tries to read or write a cassette tape, and *KVAL* is a bit that signals a received keyboard scancode when high.

The keyboard interface is little more than a simple shift register and I don't think it deserves much explanation. I only want to mention that the reading of its register, for example with the "`IN a,(2)`" instruction, also clears the *KVAL* flag.

The complete tape player computer requires the following space in the FPGA (percentages shown for a Lattice ICE40HX4K):

|  |  |  |
|---|---|---|
| Logic Cells | 2440 | 32% |
| RAM blocks | 16 | 50% |

Most of the logic cells are used by the Z80 (about 2200 LCs), the peripherals require an order of magnitude less space in the FPGA. With this occupation, there is still enough space in the FPGA for an Amstrad CPC or ZX Spectrum replica. Well, for the ZX Spectrum we are short of internal RAM mainly because we used almost all of it for the ghost-ULA and the boot ROM, so, an important redesign is needed in order to fit the tape player (moving the video memory to external RAM to start with). In the Amstrad case only two internal RAM blocks are used in the replica: one for the boot ROM (512 bytes) and another for a volume table in the AY-3-8912 (32 7-bit values), so, the tape player fits perfectly.

## 5.5 Playing .TZX (or .CDT) files and recording

Finally, it came the time to code the TZX player. The idea here is to convert the .TZX files to RLE ones inside the player in real time. These files have an internal structure consisting in an header and a sequence of variable length blocks. Each block starts with an ID byte that tells us the kind of information it carries and the way the data is encoded. And there are a lot of different blocks. Not all types are yet supported, but the most common are, allowing the loading of many .CDT files into the Amstrad replica. But the first thing we have to do is to check the file header. .TZX and .CDT files are identical and they start with a 10-byte header consisting of:

|  |  |  |  |
|---|---|---|---|
| "ZXTape!" string | 0x1A | Version | Subversion |

In our code we check the three leading bytes for the "ZXT" signature and we discard the 7 remaining bytes.

Then we can start looking for block IDs and to process each one. The currently supported block types are:

| ID | Block Type | Action |
|---|---|---|
| 0x10 | Standard Data (.TAP) | Play |
| 0x11 | Turbo Speed Data | Play |
| 0x12 | Pure Tone | Play |
| 0x13 | Pulse Sequence | Play |
| 0x14 | Pure Data | Play |
| 0x20 | Pause | Play |
| 0x21 | Group Start | Discard |
| 0x22 | Group End | Discard |
| 0x30 | Text Description | Discard |
| 0x31 | Message Info | Discard |
| 0x32 | Archive Info | Discard |
| 0x33 | Hardware Info | Discard |

Some of these blocks are simply ignored, but they have to be removed from the file data stream in order to reach the interesting ones and they still take some code for their processing (If I had my way I would put all those blocks into a single one called "Garbage"). Probably, the most common block is the "Turbo Speed Data", where the timings of the data block are specified first and then follows the data. Times are given as 16-bit values in T-state units, where a value of one is 1/3500000 of a second (The Z80 clock of a ZX spectrum 48K is the time reference). In our player the audio clock is around 44.1KHz, and therefore, these values have to be multiplied by 44.1/3500 (actually they are multiplied by (1/64 -1/512 -1/1024), that is an approximate value). But not all times are given in T-states, the duration of gaps (or pauses) are stated in milliseconds. In this case one millisecond is more or less 44 samples.

A "Turbo Speed Data" block or an "Standard Data" block will result in the generation of the following waveforms:

1. A pilot tone: A long square wave with a frequency and duration stated in the turbo speed data block or implicit in the standard data block.

2. A single sync cycle with its high and low times stated in the turbo speed data block or implicit in the standard data block. Notice that this cycle is usually a rectangular one while in order to avoid a DC component it should be an square cycle (another pair of donkey ears for ROM programmers).

3. A data block, with a duration given in bytes, where each byte results in 8 high-low cycles whose duration depends on the bits of the data. Bytes are played MSB first, and the pulse times for zero and one bits are stated in the turbo speed data block or are implicit in the standard data block. The data length is a 24-bit value for the turbo speed data block or a 16-bit value for the standard data block.

4. An inter-block gap or pause, stated in milliseconds.

A "Pure data" block is similar but lacks the pilot tone and synchronism cycle. The pilot tone could be generated with an specific block: "Pure tone", and any arbitrary sequence of pulses can also be generated using a "pulse sequence" block.

At the end, the cassette waveform is composed of a sequence of alternating high and low pulses. For instance, a zero bit in an "Standard data" block is no more than 11 samples with the cassette output high and 11 samples with the cassette line low, or in other words, two RLE bytes: $8B and $0B. Only pauses have the same cassette level on consecutive bytes. For instance, a 15 ms pause is played as 15 RLE bytes with the value $2C (44 samples low).

The current memory usage in the tape player is 6536 bytes, where 4314 bytes are for the code and the character generator table, and the rest are variables, video memory included. We still have 1260 bytes free out of a 8KB memory (1KB of memory was saved by reusing the audio buffer as storage for directory entries when selecting a file). The source code is 2174 lines long, many of them containing Z80 assembler instructions.

And also the cassette output of the Amstrad replica can now be recorded, but only to an already existing RLE file. No other formats are supported nor is the creation of new files. So, if recording is wanted please provide some long empty RLE files to overwrite (2MB should be enough for the saving of all the Amstrad memory).

## 6 Alhambra-II ports

The Alhambra-II board, when attached to its Multimedia Shield, has 8MBytes of PSRAM accessible from a Quad-SPI interface. That shield also provides a VGA interface along with audio input and output and a PS2 connector, making it a capable platform for these vintage machine clones. Memory read and writes are a bit complex and slow, but the emulated processors run with more or less a single memory access per microsecond, while the SPI clock frequency can be as high as 25MHz, thus allowing these old systems to be implemented. Take for instance an "LD A,nn" instruction, that takes 7 cycles (states) to execute in a 3.5MHz Spectrum (in the 32K memory bank) or 8 cycles in a 4MHz CPC (due to video arbitration):

| Instr. phase | states | Cycles |
|---|---|---|
| Fetch (M1) | 4 | Refresh (ignored) |
| | | Read |
| Read | 3 (4 on CPC) | Read |

Here we are performing two memory reads, that require 18 SCK cycles in the PSRAM, plus another 5 states, requiring at least a single cycle each. This results in a total of 41 SCK cycles for that instruction, or a minimum SCK frequency of 3.5MHz * 41/7 = 20.5MHz (Spectrum) or 4MHz *41/8 = 20.5MHz (Amstrad). Thus, running the quad-SPI interface with a 25MHz clock, that is a very convenient frequency due to the VGA controller, we can still have some spare time for other memory accesses, like video DMA.

The Alhambra replicas do not include tape players, even if we have an SD card connector in the shield. This connector uses the same FPGA pins as the PSRAM, and therefore it is not possible to use the SD card at the same time as the PSRAM without resorting to some complex arbitration. Also, it isn't possible to run the Z80s much faster than nominal, so, the loading of tapes is going to be done in real time. And for that purpose we already have the audio jacks, that require just a minimal amount of logic to interface (the input ADC can have a reduced resolution).

### 6.1 ZX-AlhambraMM

The block diagram of the Alhambra-MM version of the ZX spectrum is presented in figure 13. Here, along with the main Z80 CPU of the computer is a tiny BAC microcontroller. This second core is tasked with the initial configuration of the PSRAM memory and with the uploading of the Spectrum's ROM image from the Flash to the PSRAM. After this is done it releases the reset signal of the main computer and it rests in a loop blinking a LED. Using this second core there is no longer a need for a boot ROM with all its multiplexing in the Spectrum recreation.

Only the first 64KB of the PSRAM are actually in use, and the first 16KB are forced to be read-only for the Z80 processor, thus behaving like a ROM. The next 48KB can be read and written by the Z80.

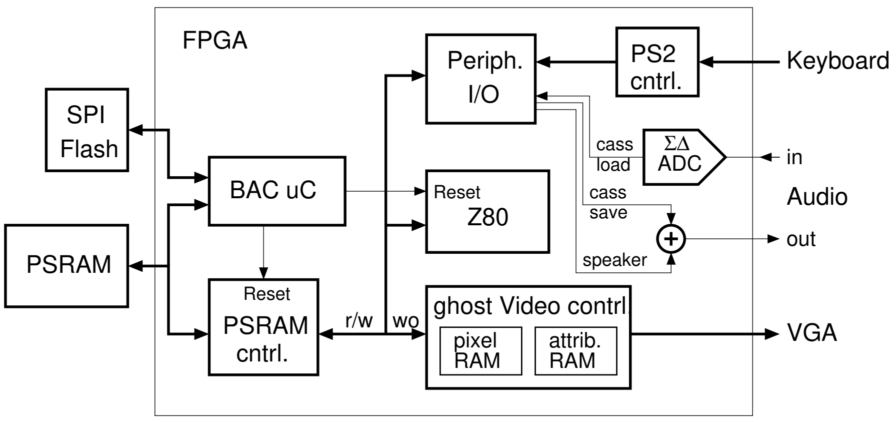


Figure 13: Block diagram of the recreated ZX Spectrum

The video controller is a little peculiar because it holds a copy of the video memory internally. This memory is written by the Z80 in parallel with the corresponding PSRAM video region but it is read only by the video refresh logic. This was possible because:

- a) There is an small amount of video RAM, just 7KB, so it fits into the internal BRAMs of the FPGA.
- b) The video memory is located always at the same address: 0x4000.

The use of a "ghost" video controller really simplifies the PSRAM controller that now only has to deal with single-byte reads and writes. In order to match more or less the speed of the original computer the video controller generates a single-cycle pulse each line, or each $32\mu s$. This pulse starts a read-write counter that gets incremented with each PSRAM read or write operation or idle cycles, and stops when reaching a count of 112. This gives an effective Z80 frequency of $112/32\mu s$ = 3.5MHz.

The timing of the PSRAM read and write cycles is presented in figure 14. Reads are 18 cycles long, comprising:

- 1 idle cycle before setting /CS low.
- 2 cycles with the read-quad command: 0xEB
- 6 cycles with the memory address (addresses are 24-bit long)
- 6 dummy cycles required by the PSRAM
- 2 cycles to read the memory byte.
- 1 idle cycle with /CS high at the end of the reading.

Writes are shorter because no dummy cycles are required in this case. Also notice that /CS is high during 2 cycles in the case of back-to-back reads / writes. This was a requirement because a 25MHz cycle last 40ns and the PSRAM requires at least 50ns with /CS high.

An state counter is used for the generation on the required control signals. This counter remains in the zero state as long as there are no read or write operations pending, that means:

- The read-write counter has reached its maximum count. Or
- The Z80 hasn't activated its /MREQ output, so this is an internal Z80 cycle. Or
- The /MREQ signal is active along the /REFRESH signal, meaning this is a Z80 refresh cycle. No refresh cycles are needed at all, but the Z80 core generates them when it is synthesized with the TV80_REFRESH switch defined. And this is required in order to have a functional R register in the Z80 core because some programs depends on it for the generation of "random" values.

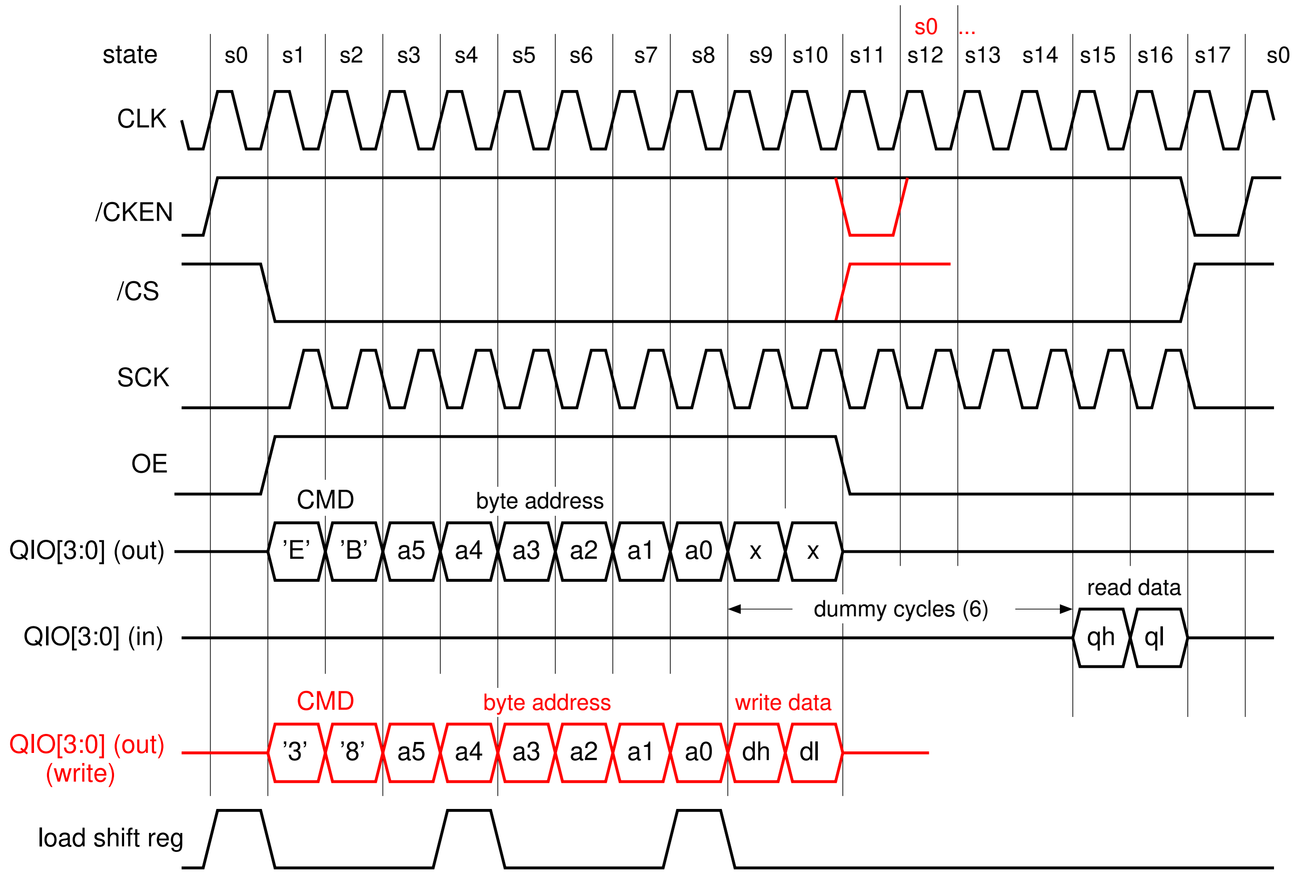


Figure 14: PSRAM timing in the recreated ZX Spectrum. Write cycles are shown in red.

The clock enable output, /CKEN, is low on the last cycle of the read / write operation. This signal is ORed with the 25MHz clock in order to obtain the CPU clock. In this way the Z80 clock, “cclk”, only gets pulses when /MREQ is inactive or the read / write operation is completed. “cclk” is also halted when the read-write counter reaches its maximum value, so, the Z80 core gets precisely 112 cycles each 32$\mu s$.

I also want to remark the following details:

- All sequential logic is updated on the rising edges of the clock. This includes the sampling of the incoming data.
- SCK is inverted with respect to the main clock.
- A 16-bit shift register is used. It shifts its contents 4-bit each time, and in consequence its data is completely shifted out after 4 cycles. Thus, the shift register is loaded 3 times:
  1. When leaving the 's0' state with the proper command on its 8 MSB (0xEB for reads or 0x38 for writes) and the 8 MSB of the memory address. (addresses are 24-bit wide)
  2. When leaving the state 's4' with the 16 LSB of the address.
  3. When leaving the state 's8' with the data to write on its 8 MSB, or with a dummy value in the case of reads.
- When reset the PSRAM controller remains in the 's0' state with '/ss' high, 'sck' low, and 'qio' in a high impedance state, so the BAC microcontroller can take over the control of the Quad-SPI bus. This reset signal is deasserted some time before the reset of the Z80 processor

Notice that this recreated Spectrum is not slowed at all when reading or writing the lower 16K RAM, so, it is going to be faster than Sinclair's, just as it was the Inves Spectrum.

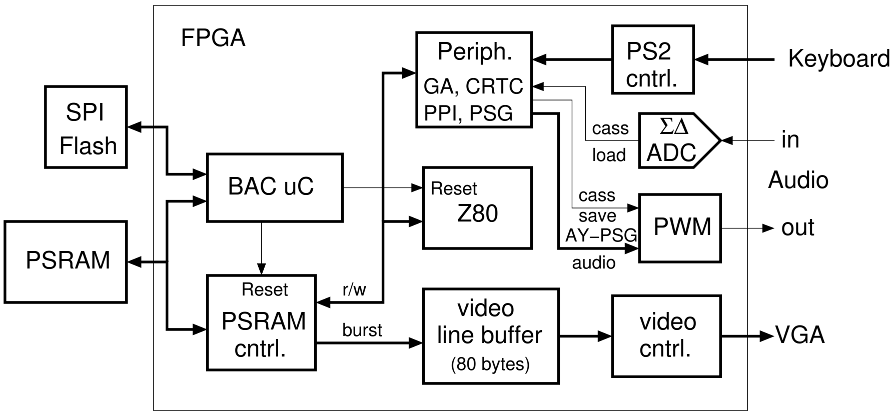


Figure 15: Block diagram of the recreated Amstrad CPC-464

After the PSRAM interface the other new blocks are the speaker and cassette peripherals. The output speaker and cassette signals are mixed together by means of a fast time-domain multiplexing. The output audio pin samples the speaker signal for 32 cycles of the main 25MHz clock while the cassette output is sampled just one cycle. This sequence is repeated each $1.32\mu s$. In this way the speaker is a loud audio signal while the cassette output is quiet but still audible.

An audio input is also included for the cassette input. In this case the sigma-delta modulator of the multimedia shield is used for the generation of a pulse train with a "density of ones" proportional to the input audio level. This ADC is capable of more than 10 bits of effective resolution, but here this is much more than really needed, so, just a 6-bit accumulator is included for the counting of the high pulses, and the MSB of the resulting count after 64 cycles is the actual cassette signal presented to the recreated Spectrum. The 5 lower bits are presented at the Alhambra LEDs as a volume meter. The ADC runs with a 1/12 of the clock frequency, or 2.08MHz, resulting in 32.5 KSamples/s.

## 6.2 CPC-AlhambraMM

The design of the CPC replica for the Alhambra-II board follows the same guidelines as the ZX Spectrum one, but there are some important differences. To begging with the biggest one, the "ghost" video controller is no longer a solution due to the amount of video RAM (16KB) and to the fact that the video base address can be changed. So, we have to refresh the screen by reading the PSRAM, and we have to modify its controller in order to allow for these video read operations.

In figure 15 the block diagram of the recreated computer is displayed, and there we can see an small memory buffer for video lines. The workings of this arrangement are:

- The video controller displays each CPC line two times in the VGA screen. Therefore a single-cycle pulse is generated every two VGA lines, or $64\mu s$, and this pulse starts the read-write counter of the PSRAM controller.

- The first five readings of the controller are bursts that store the received data into the video line buffer. A total of 80 bytes are read, 16 bytes on each burst (see the related timing in figure 16). In fact we could read all the 80 bytes in a single burst, but it is done this way in order to avoid reading at wrong addresses during video lines with address wraparounds. (This can happen when scrolling text on the screen).

- The video controller refresh the screen using the data stored on the buffer, that is read twice on two VGA lines before updating its contents. The buffer is implemented using just a single BRAM.

During the $64\mu s$ interval we have the five video read bursts followed by the emulated CPU cycles. And here there is some uncertainty: A 4MHz CPU will receive 256 clock pulses during this time, but the effective

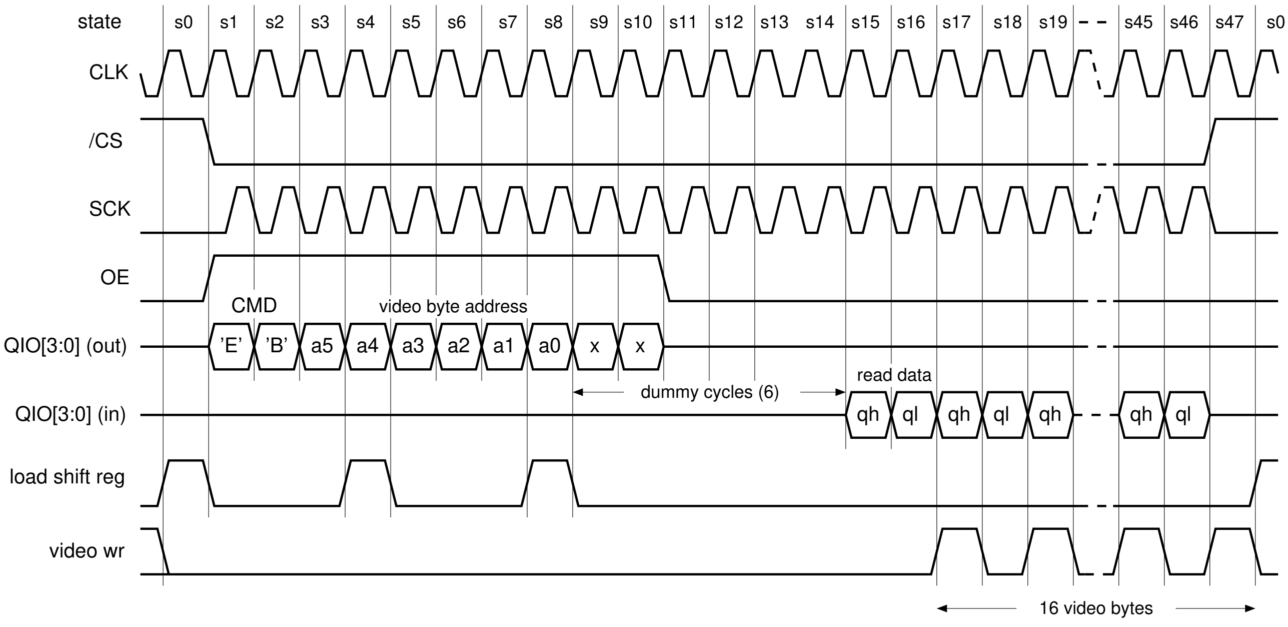


Figure 16: PSRAM timing during video read bursts in the CPC replica.

clock frequency of the Amstrad CPC is lower due to the wait states inserted for video arbitration. In Wikipedia a 3.3MHz effective frequency is stated after referencing Amstrad sources, while for the particular case of the "LD A,nn" instruction this frequency is a bit higher: 3.5MHz, the same value we get in the ZX Spectrum. Any value from 3.3MHz to 4MHz is easy to get in the emulated machine as there is enough time even for the maximum clock speed. But at the end I'm keeping the same frequency of the Spectrum, or 3.5MHz, that results in 224 CPU cycles for the $64\mu s$ interval.

There are, of course, many other differences with respect to the Spectrum, starting with the peripheral set of the CPC, that includes an specific Gate Array, a MC6845 CRT Controller, an 8255 Parallel Peripheral Interface, and a General Instrument's AY-3-8912 sound chip. The functionality of these peripherals had to be minimally replicated in order to have a working CPC, while in comparison the ZX only has its single ULA port.

The audio generation also has another notable difference: The output of the recreated PSG is a digital sample with 8 bits of resolution. To these samples the value of the cassette output pin is added as a two-level signal with an amplitude of 15 LSBs peak to peak, and the mixed signal is fed into a pulse width modulator that results in an analog audio wave after the low-pass filter of the multimedia shield. With this arrangement we can ear both the tunes of the PSG and the (attenuated) saving of programs into "cassette tapes".

And talking about cassettes, the CPC provides an output to control the motor of the cassette player that has no use in the replica. Yet, its value is displayed into the lower LED of the Alhambra board.

# 7 Annexes

## 7.1 Amstrad CPC TAPE FORMAT

The data cassettes used in the Amstrad CPC have the format described here (At least those tapes recorded using the "SAVE" command of its Locomotive BASIC). Most of this information was obtained by reverse engineering tape images.

The format is that shown in figure 17, where we can see how a file is first split into several blocks. These blocks can carry up to 2KB of data each, the last one usually being smaller. Blocks are separated by more than 2 seconds of silence in the tape and they also have an inner structure. Basically one block includes two cassette

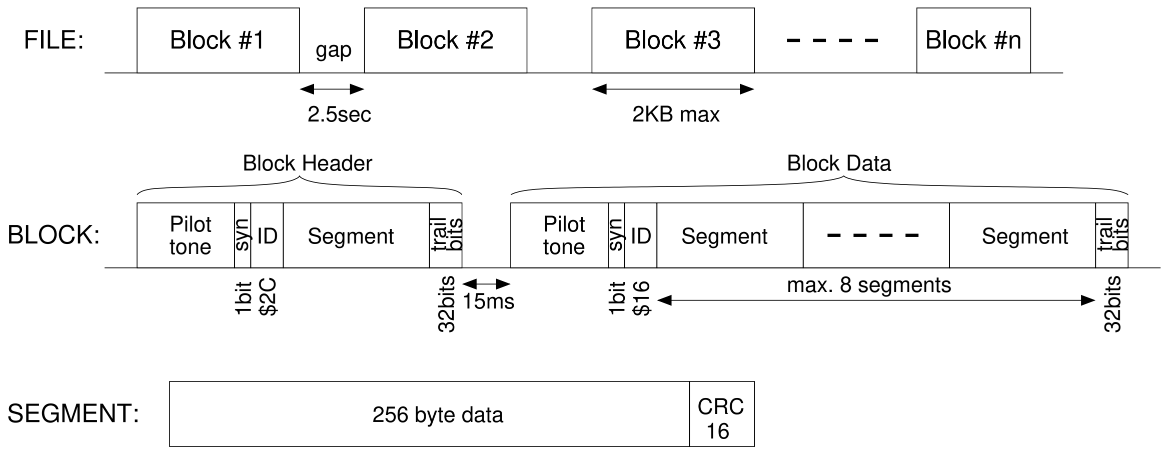


Figure 17: Data structure of Amstrad CPC tapes

bursts of audio separated by an small gap of only 15ms of silence. The first one is the block header and it is followed by the data of the block. The two waveforms start with a pilot tone that comprises 2048 bits in one followed by a single synchronism bit in zero. Then comes a byte with an identifier. This byte has the value $2C for the block header or $16 for the block data.

After the ID byte comes one or more segments of data. These segments are always 258 bytes long, where 256 bytes are the actual data and 2 more bytes are added with the CRC-16 of the previous 256 bytes. The header has a single segment while the data part of the block can be up to 8 segment long. If a segment has less than 256 bytes of data it will be filled with zero bytes up to its nominal 256 byte length.

After the last segment of the block header, or block data, there are another 32 trailing one bits before the silence of the following gap.

Bytes are saved MSB first, and the CRC-16 value is also saved MSB first (big endian). Each individual bit is coded as a single cycle square wave: a short wave for zeroes, and a long wave for ones, resulting in a variable data rate. Times aren't required to be very accurate, mainly because of cassette variations in playing speed, but typically a zero last about 325$\mu s$ high and 325$\mu s$ low and a one last double the time (650$\mu s$ high and low). This results in an average data rate about 1Kbit per second (The data rate actually depends on the percentage of bits in zero or one in the data. This figure is for a 50% bits of each type).

The CRC calculation is also a little peculiar because the CRC starts with all bits in one instead of zero, and the resulting CRC is inverted before adding it to the tape. The CRC polynomial is the usual $x^{16} + x^{12} + x^5 + 1$, (or conversely, its XOR value is 0x1021). The code for the calculation of this CRC is the following:

```
unsigned short cpc_crc(unsigned char *buf,unsigned int n)
{
    unsigned int i;
    unsigned short crc;
    crc=0xFFFF;                // Init with all ones
    for(i=0;i<n;i++)
      {
        crc ^= (buf[i] << 8); // move byte into MSB of 16bit CRC
        for (int i = 0; i < 8; i++)
        {
            if (crc & 0x8000) crc = ((crc << 1) ^ 0x1021);
            else crc <<= 1;
        }
    }
```

```
        return ~crc; // return value complemented
    }
```

This kind of CRC seems to be also used today by GENIBUS communications (Grundfos water pumps and related)

Finally, we have to describe the contents of the segment of the block header. Here only the first bytes of the segment are used and they are:

| Offset (bytes) | Size (bytes) | Field | Comments |
|---|---|---|---|
| 0 | 16 | File name | ASCII string |
| 16 | 1 | Block number | First block is #1 |
| 17 | 1 | End block flag | $FF if this is the last block of the file, $00 otherwise |
| 18 | 1 | File type | See table below |
| 19 | 2 | Block length | Length of useful data (real length is multiple of 256) |
| 21 | 2 | Destination address | Where to write this block |
| 23 | 1 | First block flag | $FF if this is the first block of the file, $00 otherwise |
| 24 | 2 | Total file length | |
| 26 | 2 | Entry address | Address to jump for a *RUN ""* command |
| 28 | 228 | Unused | all $00 |

Also, notice that the fields "File Name", "File Type", "Total File Length", and "Entry Address" are the same in all the blocks of the file.

The byte "File type" has the following values:

| Value | File type |
|---|---|
| $00 | BASIC program |
| $01 | Protected BASIC program |
| $02 | Binary data |
| $04 | Screen dump (not sure) |
| $16 | ASCII file |

Protected programs can't be listed but they can be executed.

In overall the tape format of the Amstrad CPC is probably one of the best of its time, even including a sophisticated CRC for error detection. I only want to point out two aspects of it I dislike:

- The use of FM coding that results in different times for zeroes and ones. In that respect a Differential Manchester coding, like the one used in Thomson computers, gives a constant data rate and a 50% more data for the same bandwidth (but Thomsons used a weaker checksum instead of CRC). Still, these codings are far form the performance of Wozniak's Group Coded Recording.

- The long gaps and pilot tones are a waste of time, so, the splitting of files into a lot of small blocks is really a bad idea. It should be better to allow more than 8 segments for each block and to reduce the number of blocks to a single one, something that certain games already do in their loaders.

## 7.2 .PSG player source code (CPC464)

```
        output "psg.bin"

        org     $1200
play:   di                              ; Interrupts will mess with PPI due to keyboard
        ld      bc,$f782                ; Set PPI PORTs: A output, B input, C output
        out     (c),c
        ld      hl,musica               ; pointer to PSG file
pl00:   ld      de,musend               ; if (EOF) return
        ex      de,hl
        ccf
        sbc     hl,de
        ex      de,hl
        jr      nc,pl01
        ei
        ret
pl01:   ld      a,(hl)                  ; A=*HL++;
        inc     hl
        cp      $FE
        jr      nz,pl0
        ld      a,(hl)                  ; if (A==0xFE) C=delay=4*(*HL++)
        inc     hl
        add     a
        add     a
        ld      c,a
        jr      pl10
pl0:    cp      $FF
        jr      nz,pl2
        ld      c,1                     ; if (A==0xFF) C=delay=1
pl10:   ld      b,$f5                   ; wait for n VSYN falling edges (n in C reg)
pl11:   in      a,(c)                   ; while (PPI.PORTB.0 == 0)
        and     1
        jr      z,pl11
pl12:   in      a,(c)                   ; while (PPI.PORTB.0 != 0)
        and     1
        jr      nz,pl12
        dec     c
        jr      nz,pl11
        jr      pl00                    ; continue

pl2:    ld      c,a                     ; register in C
        ld      a,(hl)                  ; A=value=*HL++
        inc     hl
write_to_psg:
        ld      b,$f4           ; setup PSG register number on PPI port A
        out     (c),c           ;
        ld      bc,$f6c0        ; Tell PSG to select register from data on PPI port A
        out     (c),c           ;
        ld      bc,$f600        ; Put PSG into inactive state.
        out     (c),c           ;
        ld      b,$f4           ; setup register data on PPI port A
        out     (c),a           ;
        ld      bc,$f680        ; Tell PSG to write data on PPI port A into selected register
        out     (c),c           ;
        ld      bc,$f600        ; Put PSG into inactive state
        out     (c),c           ;
        jr      pl00            ; continue

;musica:  ; db table with PSG file contents
 include "musica.inc"
musend:
```