INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

Realização de um ZX Spectrum em FPGA

Gustavo Dinis Venturinha Cercas Lopes Jacinto

(Licenciado)

Dissertação de natureza científica para obtenção do grau de Mestre em Engenharia Informática e de Computadores

Orientador : Professor Doutor Rui Policarpo Duarte

Júri:

Presidente: Professor Doutor Tiago Miguel Braga da Silva Dias

Vogais: Professor Doutor Alberto Cunha
Professor Doutor Rui Policarpo Duarte

September, 2023

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

Realização de um ZX Spectrum em FPGA

Gustavo Dinis Venturinha Cercas Lopes Jacinto

(Licenciado)

Dissertação de natureza científica para obtenção do grau de Mestre em Engenharia Informática e de Computadores

Orientador : Professor Doutor Rui Policarpo Duarte

Júri:

Presidente: Professor Doutor Tiago Miguel Braga da Silva Dias

Vogais: Professor Doutor Alberto Cunha
Professor Doutor Rui Policarpo Duarte

September, 2023

Acknowledgments

I would like to thank my advisor for his tremendous support and feedback, as well as his patience in teaching me how to use the software and materials for this project, and proofreading the dissertation, despite all the other projects he was involved in at the time.

Thanks to my classmates for their input about the dissertation and overall support, as well as the fun we had hanging out, which served as good breathers from the project.

I would also like to thank my parents for believing in me and supporting me throughout this work's execution.

Abstract

The ZX Spectrum was a popular 8-bit home computer by Sinclair Research from the 80s. The peripherals used by the ZX Spectrum, such as analog television and audio cassette tapes, nowadays are outdated. Thus, these computers are not easily usable today. With this in mind, an upgrade to the ZX Spectrum 48k could allow support for modern peripherals while keeping compatibility with the original ZX Spectrum 48k. In this research project, a Terasic DE2-115 FPGA board was chosen as the target platform to implement the ZX Spectrum with its novel peripheral support. This board includes an Intel Cyclone IV FPGA, a VGA port, a PS/2 port, an analog audio I/O, an SD card slot, and an I/O expansion for connecting joysticks. The work in this dissertation involved the implementation of the complete original system in reconfigurable hardware, as well as all the controllers for the new peripherals, in VHDL. Besides the hardware, software was also developed in assembly for the Z80 CPU. A NIOS II soft processor was included and programmed to provide novel functionality to an old computer system that it could not support otherwise. This work demonstrates the possibility of reutilizing architectures like the proposed one to update old systems. Results show that the modernized architecture functions like the original one. The complete modernized ZX Spectrum project is publicly available on Git Hub as open-source.

Keywords: FPGA, NIOS II, retro-computing, Z80, ZX Spectrum, VHDL, Assembly

Resumo

O ZX Spectrum era um computador doméstico popular de 8 bits da Sinclair Research dos anos 80. Os periféricos utilizados pelo ZX Spectrum, como a televisão analógica e as cassetes de áudio, hoje em dia estão desactualizados. Assim, estes computadores não são facilmente utilizáveis hoje em dia. Tendo isto em conta, uma atualização para o ZX Spectrum 48k poderia permitir o suporte de periféricos modernos, mantendo a compatibilidade com o ZX Spectrum 48k original. Neste projeto de investigação, foi escolhida uma placa FPGA Terasic DE2-115 como plataforma alvo para implementar o ZX Spectrum 48k com o seu suporte de novos periféricos. Esta placa inclui uma FPGA Intel Cyclone IV, uma porta VGA, uma porta PS/2, uma I/O de áudio analógica, uma ranhura para cartões SD e uma expansão de I/O para ligar joysticks. O trabalho desta dissertação envolveu a implementação do sistema original completo em hardware reconfigurável, bem como de todos os controladores para os novos periféricos, em VHDL. Para além do hardware, foi também desenvolvido software em assembly para o CPU Z80. Um processador de software NIOS II foi incluído e programado para fornecer novas funcionalidades a um sistema informático antigo que este não poderia suportar de outra forma. Este trabalho demonstra a possibilidade de reutilizar arquitecturas, como a proposta, para atualizar sistemas antigos. Os resultados mostram que a arquitetura modernizada funciona como a original. O projeto completo do ZX Spectrum modernizado está disponível publicamente no Git Hub como código aberto.

Palavras-chave: FPGA, NIOS II, computação retro, Z80, ZX Spectrum, VHDL, Assembly

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ADC Analog-to-Digital Converter. 41

BRAM Block RAM. 32, 34, 49, 65

CHS Cylinder-Head-Sector. 88
CLB Configurable Logic Block. 38
CODEC Coder/Decoder. 55, 56, 75, 79
CPU Central Processing Unit. vii, xx, 2, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 19, 20, 27, 28, 29, 30, 31, 32, 33, 36, 42, 43, 48, 53, 55, 57, 59, 61, 63, 83, 84, 85, 90, 99, 112, 113, 114, 118, 120, 121, 122
CRC Cyclic Redundancy Check. 43
CS Chip Select. 43

DAC Digital-to-Analog Converter. 20, 34, 39, 41, 49, 50, 55, 57, 79
DMA Direct Memory Access. 8, 14, 53, 62, 83, 84, 91, 92, 101
DRAM Dynamic RAM. 7, 8
DSP Digital Signal Processors. 34, 38, 106

FAT File Allocation Table. 88
FPGA Field-Programmable Gate Array. vii, 1, 2, 27, 28, 29, 30, 31, 32, 34, 36, 37, 38, 39, 41, 42, 43, 45, 49, 52, 59, 75, 88, 106, 110, 125, 126

GAL Generic Array Logic. 45

HDL Hardware Description Language. 37
HDMI High-Definition Multimedia Interface. 31, 32, 125

I/O Input/Output. vii, 5, 7, 8, 10, 12, 14, 15, 16, 18, 20, 21, 25, 32, 37, 38, 43, 45, 54, 61, 62, 82, 84, 85, 86, 108, 117, 120, 125, 126
I2C Inter-Integrated Circuit. 41
IC Integrated Circuit. 5
IEC International Electrotechnical Commission. 32, 34
IEEE the Institute of Electrical and Electronics Engineers. 28
IFF Interrupt enable Flip-Flop. 8, 52, 90
IP Intellectual Property. 34, 51, 56, 59, 62, 81
ISO International Organization for Standardization. 32, 34
IST Instituto Superior Técnico. 33

LAN Local Area Network. 23
LBA Logic Block Address. 88
LE Logic Element. 38, 45, 106
LSB Least-Significant Bit. 81
LUT Look-up Table. 38

MSB Most-Significant Bit. 55, 56, 79, 83

NES Nintendo Entertainment System. 32, 82
NMI Non-Maskable Interrupt. 8, 53, 56, 57, 84, 90, 94, 101, 102, 103

PAL Programmable Array Logic. 45
PC Program Counter. 8, 9, 56, 90, 91, 92, 94, 101
PCB Printed Circuit Board. 2, 45, 75
PLD Programmable Logic Device. 37, 45
PLL Phase-Locked Loop. 28, 32, 34, 38, 60, 83, 106
PROM Programmable ROM. 37

RAM Random-Access Memory. xxv, 5, 6, 7, 11, 13, 15, 29, 32, 33, 49, 50, 53, 54, 55, 60, 61, 62, 91, 96, 102, 103, 108, 117

RGB Red, Green, and Blue. 15, 27, 31, 39, 66

ROM Read-Only Memory. xxiii, 6, 11, 21, 23, 52, 53, 54, 60, 94, 96, 97, 98, 99, 102, 103, 108, 112, 114, 117

SP Stack Pointer. 9, 56, 92, 101, 102

SPI Serial Peripheral Interface. 43, 88

ULA Uncommited Logic Array. 5, 6, 10, 11, 12, 13, 14, 15, 16, 18, 19, 20, 28, 29, 30, 31, 32, 33, 45, 47, 48, 50, 51, 52, 53, 55, 56, 57, 60, 61, 62, 63, 64, 65, 69, 79, 83, 89, 90, 114, 118, 120, 121

USB Universal Serial Bus. 32

VGA Video Graphics Array. vii, 1, 2, 27, 28, 31, 32, 34, 37, 39, 41, 49, 50, 57, 64, 65, 67, 69, 70, 125

VHDL VHSIC Hardware Description Language. vii, 27, 28, 29, 31, 32, 33, 37, 48, 57, 59, 65, 75, 102

VHSIC Very High Speed Integrated Circuit. 27

1

Introduction

When the ZX Spectrum was released, it was a breakthrough in personal computing, accessible for mass marketing. Its pricing, as well as the use of analog TV and audio cassettes, made it wildly accessible to the public at that time. This report describes a research project to design and implement the hardware for a modern ZX Spectrum using Field-Programmable Gate Array (FPGA) technology and novel peripherals.

This chapter contains the motivation behind this project, followed by its objectives and the document's outline.

1.1 Motivation

This project's goal is to recreate a ZX Spectrum computer on a modern hardware platform. This implementation interfaces the original system with modern peripherals, such as VGA monitors for video output and SD cards for loading programs, surpassing the limitation of having to use audio cassettes and analog television. This project focuses on a hardware design targeting an Field-Programmable Gate Array (FPGA) board that uses more peripherals when compared to previous works, as well as interfacing with Joysticks and an original ZX Spectrum keyboard.

Many ZX Spectrum emulators have been done for different platforms, in software and hardware. However, software emulators tend to have behaviors or phenomena, caused by the underlying operating system, that do not exist in the original ZX Spectrum.

Since these implementations run on top of a system, they are limited to its best performance and latency which may not be enough to faithfully replicate the response of the software running on the Spectrum.

The main reason for a hardware implementation rather than a software one was due to achieving the maximum fidelity with the original ZX Spectrum. Many hardware implementations of the old computer were developed by retro gaming communities in ways that are difficult/impossible to reproduce and imply the assembly of custom Printed Circuit Board (PCB)s that may not be accessible. FPGA technology was chosen due to its easy reconfigurability and testability, as well as integration and customization.The use of an FPGA development kit allows a hardware design to be created without the need to design and assemble a custom and complex PCB. The motivation behind analyzing similar projects is to reuse already validated IP cores, like the Z80 CPU.

1.2 Objectives

This project's objective is to recreate a ZX Spectrum on a modern hardware platform. The developed system should be able to load and play games as close to the original hardware as possible. This means having the Z80 CPU implemented and all the peripherals working like the original ZX Spectrum. The proposed system has to support the following peripherals:

- **VGA** for video output, used by common computer monitors;
- **PS/2** for keyboard input and the original **ZX Spectrum's keyboard** (optional);
- **SD card** for faster loading programs and games;
- **Audio in** for loading audio cassette tapes, allowing old games to be played;
- **Audio output** for sounds and music;
- **Kempston and Sinclair joystick** input.

1.3 Dissertation Outline

This report is organized as follows:

- Chapter 2: **The ZX Spectrum** explains the original system and its components;

- Chapter 3: **Related Work** introduces and analyzes similar projects that are rated based on the needs of this project;
- Chapter 4: **The Target Platform** explain the platform this implementation was designed on;
- Chapter 5: **Proposed Modernized ZX Spectrum Fusion Architecture** presents the proposed architecture for the implementation, along with design choices that were made;
- Chapter 6: **Implementation of the ZX Spectrum Fusion** details the implementation of each aspect of the machine, mainly the components that were created and how they connect;
- Chapter 7: **Unity Tests and Evaluation** contains the methods used to test the components and the project as a whole.

1.4 Contributions

During the execution of this work, the following contributions were made:

1. A paper was submitted to the Electronics journal from MDPI (ISSN 2079-9292);
2. The project's repository was made available on GitHub (https://github.com/A46006/ZX-Fusion).

2

The ZX Spectrum

This chapter presents a background on the ZX Spectrum+ 48k computer. It explains its multiple components: the Z80 CPU, the Uncommited Logic Array (ULA), the video output and organization, the keyboard, and the audio Input/Output (I/O).

The ZX Spectrum is an 8-bit home computer developed by Sinclair Research and it was released released in 1982. The first versions released had either 16KB or 48KB of Random-Access Memory (RAM), named the ZX Spectrum 16K/48K. Later, other versions came out with more RAM and improvements to the keyboard (ZX Spectrum 128K and ZX Spectrum+, respectively). The British electronics company Amstrad, after purchasing the Sinclair brand and the ZX Spectrum range, released the ZX Spectrum +2, which featured a built-in audio cassette recorder and a spring-loaded keyboard, much like Macintosh and PC keyboards. The ZX Spectrum +3 featured a built-in floppy disk drive as well as core changes that brought compatibility issues with certain games and interfaces that worked in previous models.

The ZX Spectrum, in general, is a home computer in the shape of a keyboard that can be plugged into an analog television set to use as a screen, much like its competitors, the Commodore 64 and the MSX, and its predecessors the ZX80 and the ZX81. In these computers, software was loaded/saved from/to audio cassette tapes through the audio I/O ports, since audio cassettes were the most common and cheap audio media at that time and easy to interface.

The motherboard of a ZX Spectrum+ 48k model is shown in Figure 2.1. This circuit board contains the following Integrated Circuits (ICs)[56]:

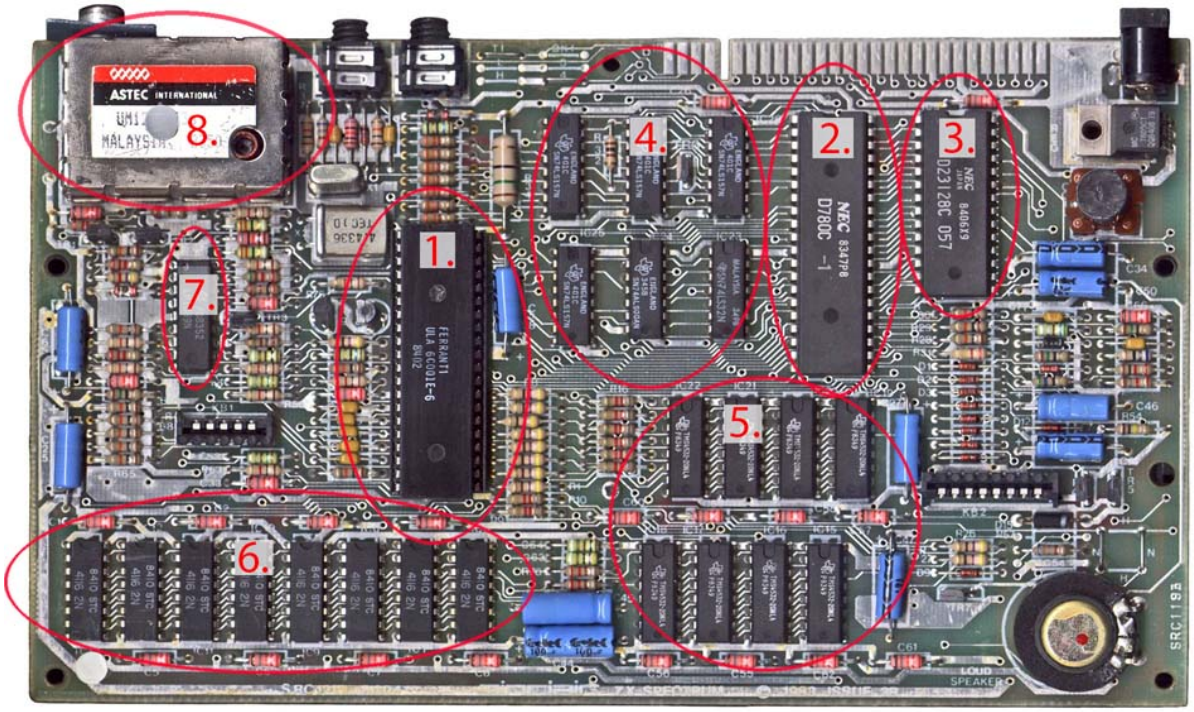

Figure 2.1: Motherboard of ZX Spectrum 48k models[56]

1. The ULA, explained in section 2.3;

2. The Z80 CPU, explained in section 2.1;

3. The Read-Only Memory (ROM) chip, which holds the ZX Spectrum's firmware;

4. Decoder/Multiplexer chips, used for calculating row and column values for the RAM based on addresses fed to them by the CPU and/or ULA;

5. Upper RAM, only accessible to the CPU;

6. Lower RAM, accessible to the CPU and the ULA;

7. Video modulator;

8. Astec UM1233 RF Modulator, which takes video and audio signals and turns them into an analog TV channel.

The next sub-sections dive deeper into the components and inner workings of the original ZX Spectrum.

2.1 The Z80 CPU

The CPU present in ZX Spectrum computers is the Zilog Z80, an 8-bit microprocessor with 16-bit memory addressing support. It is an extension and enhancement of the Intel 8080, led by Federico Faggin when he worked at Intel. He conceived the Z80 in late 1974 [22], and it was aimed mainly at embedded systems. This CPU was widely used by home computers and consoles at the time like the MSX, the ColecoVision, Sega's Mega Drive, and Nintendo's Game Boy, and is still used in Texas Instrument graphing calculators and other low-cost devices to this day [45]. One of its big features is the capability of supporting the refreshing of Dynamic RAM (RAM) on its own, removing the necessity for additional DRAM controllers

The I/Os of the Z80 CPUs are in Figure 2.2. This CPU possesses separate address and data bus lines, unlike many CPUs of that era which would multiplex these signals through the same bus.

The CPU control signals serve to indicate to the rest of the system what it is currently doing, either a memory access or an I/O access, a read, or a write. Moreover, there are control signals to indicate if the 7 lowest bits of the address can be used as a refresh signal for DRAM (using the RFSH signal) and if the current machine cycle is the opcode fetch (M1) or, in conjunction with IORQ, if the current cycle is an interrupt acknowledge [21].

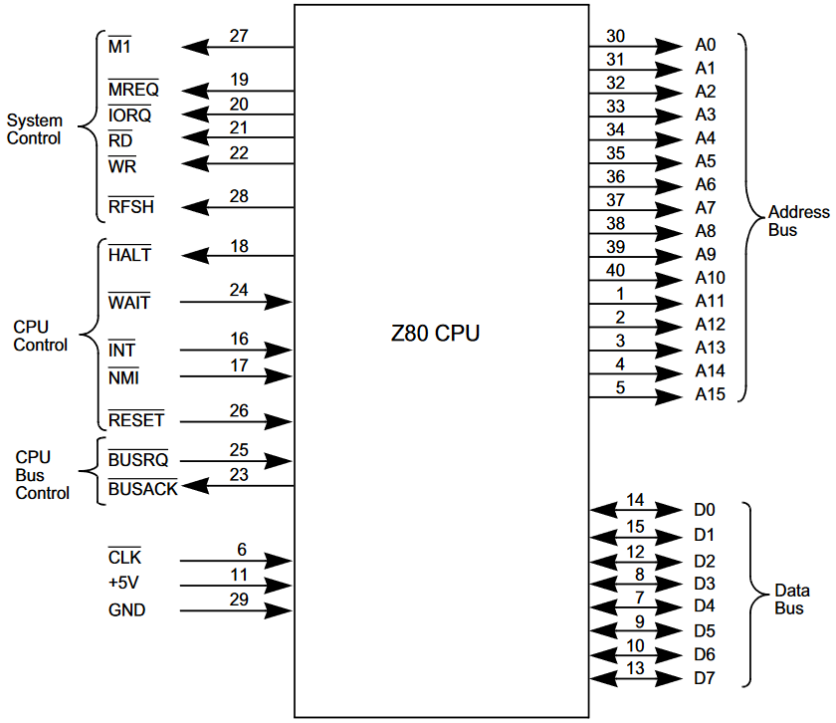

Figure 2.2: The Z80's pins, acording to the manufacturer's datasheet [21]

From the CPU control group of pins, their functions are as follows[21]:

- **HALT** informs that the CPU has executed a HALT instruction and awaits an interrupt before it can resume. In this state, it can still refresh DRAM;
- **WAIT** tells the CPU that addressed memory or I/O devices are not ready for data transfer. Extended WAITs can prevent the refresh of DRAM;
- **INT** and **NMI** for triggering external interrupts, maskable and non-maskable respectively. A Non-Maskable Interrupt (NMI) has higher priority due to being independent from the Interrupt enable Flip-Flop (IFF) inside the Z80, which save the state of maskable interrupts (on/off). The NMI forces the CPU to execute code at address 0x0066;
- **RESET** initializes the CPU. This clears the Program Counter (PC), the I and R registers, the interrupt enable flip-flops, and the Interrupt Mode.

The Z80 Bus Control pins are used for other devices to have access to the address and data busses, as well as System Control signals, by having them in high impedance. **BUSRQ** is used to request access to these signals and **BUSACK** informs that these signals are at high impedance[21]. This allows peripherals to have Direct Memory Access (DMA).

The CPU can be programmed to respond to the maskable interrupt in one of these three different modes[21]:

- **Mode 0** allows the interrupting device to feed instructions directly to the Z80. These instructions require more clock cycles due to the CPU's addition of two wait states before an Interrupt response cycle. This is done to make time for the implementation of an external daisy chain for priority control;
- **Mode 1** activates a vectored interrupt that pushes the current PC to stack and sets PC = 0x0038;
- **Mode 2** is also a vectored interrupt, enabling the user to make an indirect call to any memory location by supplying a single 8-bit value. The programmer maintains a table of 16-bit starting addresses for every interrupt service routine and, when an interrupt is accepted, a 16-bit pointer must be formed to obtain one of these starting addresses from the table. The upper 8 bits of this pointer are to be stored in the contents of the I register with an instruction such as LD (Load). The least significant bit of the pointer must be a 0, a requirement caused by the need for the CPU to fetch two adjacent bytes to form a 16-bit address and these must always start at even addresses.

The Z80 contains eighteen 8-bit registers and four 16-bit registers. It also possesses two sets of accumulator and flag registers and six special-purpose registers, including the PC and the Stack Pointer (SP). These two are 16-bit wide, as well as the index registers IX and IY, which are used in indexed addressing modes[21].

The Z80 instruction set consists of 78 instructions from the Intel 8080A, and 80 novel instructions. The instructions fall into various categories: 'Load and Exchange', 'Block Transfer and Search', 'Arithmetic and Logical', 'Rotate and Shift', 'Bit Manipulation', 'Jump, Call and Return', 'Input/Output' and 'Basic CPU Control'[21].

Addressing modes supported by this CPU include:

- Immediate Addressing;
- Immediate Extended Addressing, which receives a 2-byte operand for a 16-bit value;
- Modified Page Zero Addressing, that uses a single byte CALL instruction that sets the PC to an effective address in page zero of memory;
- Relative Addressing;

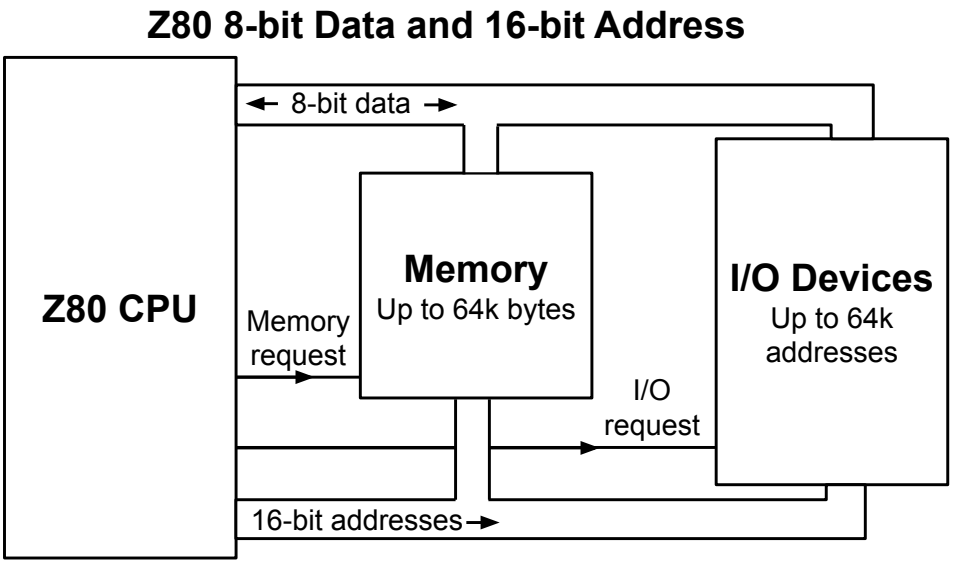

Figure 2.3: Z80's interaction with memory and I/O in most computers it is used in [44]

- Extended Addressing;
- Indexed Addressing, which receives a displacement value that is added to one of the index registers, without altering it, to form a pointer to memory;
- Register Addressing;
- Implied Addressing, instructions that imply the presence of operands in specific CPU registers;
- Register Indirect Addressing, which specifies a 16-bit register pair to be used as a pointer;
- Bit Addressing.

These addressing modes can be combined in instructions that require more than one operand.[21]

In the ZX Spectrum, the Z80 CPU uses a clock of 3.5MHz and, unlike Figure 2.3 shows, does not interact directly with all the I/O devices, specifically the main ones such as the keyboard and TV. The ULA is responsible for interfacing with these devices and the CPU communicates with it to send/receive data between these devices. This interaction is detailed below, in Section 2.3.

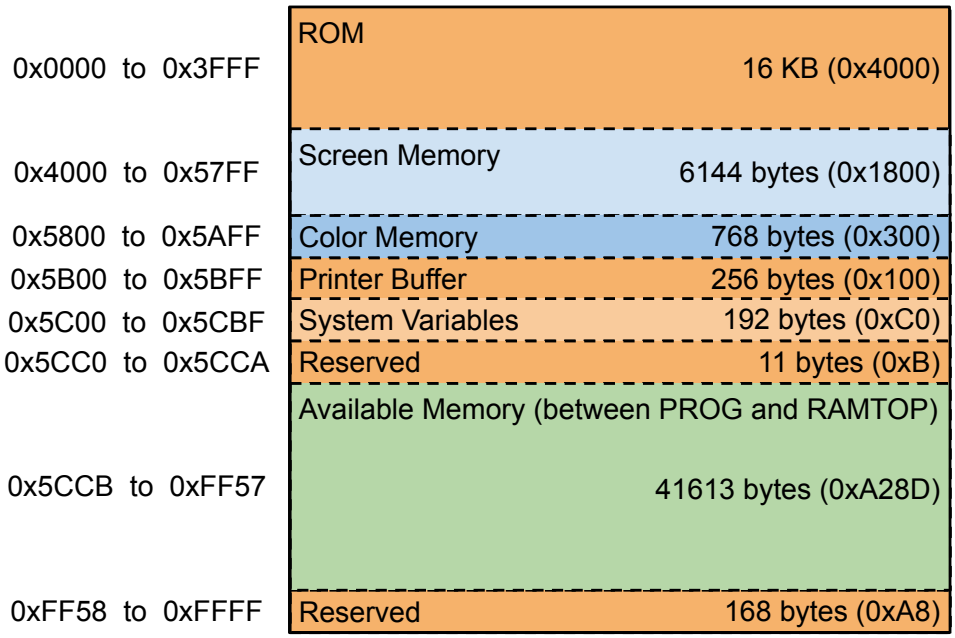

Figure 2.4: ZX Spectrum's memory map[54]

2.2 Memory

The memory addressing space of the ZX Spectrum 48k is in Figure 2.4. The low 16KB are used for the ROM whilst the rest of the address space is for RAM. Starting at address 0x386E of ROM, there are 1170 bytes of unused space, filled in by 0xFF values. The lowest blocks of RAM are reserved for screen memory, accessed by the ULA to read the video data. The memory available to programmers starts at 0x5CCB and ends at 0xFF58, but time-sensitive code could only be placed in addresses higher than 0x8000 due to the memory contention between the CPU and the ULA in lower RAM, detailed below in Subsection 2.3.4.

2.3 ULA

2.3.1 History

An ULA, also known as an Uncommited Logic Array, is a semiconductor technology that was used to design and produce custom logic chips launched in 1983. The commercialization of it was pioneered by Ferranti in the UK, offering circuits of "100 to 10,000 gates and above". [14] It is comprised of various logic building blocks which are later connected to support a custom logic circuit design during the manufacturing

Figure 2.5: A ZX80's board (left) and a ZX81's board (right)[61]

process according to the intentions of the customer [18]. In terms of innovation, Sinclair Research used the ULA to create the ZX Spectrum's predecessor, the ZX81 (Mar. 1981), based on its predecessor, the ZX80 (Jan. 1980). These computers were technically very similar, but the use of an ULA in the ZX81 allowed the discrete logic of the ZX80, originally implemented using multiple discrete 74LSxx chips, to be integrated into a single chip [62]. This change lowered the chip count from 21 to 4, allowing Sinclair to greatly reduce the price of the ZX81 when compared to the ZX80 due to its simpler construction. The difference between the two can be seen in Figure 2.5, which shows the number of chips used on each one.

2.3.2 Interfacing Peripherals

In the ZX Spectrum, the ULA is responsible for interfacing the Z80 with peripherals such as the keyboard, video output, and audio in/out, much like the ZX81.[19]

The Z80 interacts with the ULA as a peripheral by accessing I/O port 0xFE. By reading this port, the CPU obtains the signal data being received in the EAR input from an audio cassette player. It also receives keyboard data to determine key presses based on the highest 8 bits of the address line. Further details about the keyboard data can be found in Section 2.5. The data bus organization for this read can be seen in Table 2.1, in the first row. This table also shows the bits on the data bus when the CPU writes to port

0xFE. This operation is used to generate the MIC and EAR outputs by affecting bits 3 and 4 respectively. The lowest 3 bits are used to set the color of the screen's border. [4] This color is kept in a register, in the ULA, that directly feeds the display controller for ease of access.[27](page 212)

Table 2.1: The data bus contents when the CPU accesses IO address 0xFE

Operation	7	6	5	4	3	2	1	0
Read	-	EAR	-	Keyboard				
Write	-	-	-	Spkr	MIC	Border		

2.3.3 Video Counters

The ZX Spectrum's ULA uses a crystal oscillator of 14MHz, which it divides in two for a 7 MHz clock signal, with a guaranteed 50% duty cycle, used to synchronize all of its synchronous components. It holds a horizontal counter, a vertical counter, and a flash counter.

The horizontal and vertical counters keep track of the television's scanning position and are used to calculate the current address of the screen data in memory to be displayed. When the television is in the blank region at the bottom of the screen, an interrupt (Mode 1) is triggered on the Z80, which triggers the execution of the routine that increments the spectrum's frame counter, stored in RAM, and then scans the keyboard.

The horizontal counter has two stages, the first one is a free-running 6-bit ripple counter with negative edge triggered D-type flip-flops, and the second is a 3-bit synchronous T-type flip-flop (with reset, carry, and enable) counter, also negative edge triggered. The vertical counter is a full 9-bit synchronous counter of T-type flip-flops with carry and enable.

There is also a flash counter, used to obtain the frequency of changing colors in flashing attribute blocks (check Section 2.4). It counts 32 display frames with a 5-stage D-type flip-flop ripple counter, clocked with the vertical counter's last bit (which goes low once per frame). This means that for the frame rate of 50.0801 Hz, the frequency of the counter's last bit is 1.565 Hz, which corresponds to one flash per 0.639 seconds.[27](page 135)

2.3.4 Lower Memory Contention

Lower Memory Contention refers to the lower RAM (16-32K) being accessible by both the Z80 and the ULA. These two, if not controlled in relation to each other, could access

the memory simultaneously and corrupt it. For instance, the ULA could be reading video data for the display whilst the Z80 accesses data in that region for another purpose. To prevent this, the Z80's clock derives from the ULA, allowing the ULA control over its operating time if any upcoming data access to that region is detected while the ULA is reading video data. If an access is detected, the CPU's clock signal is held high while the ULA finishes its read. The ULA has priority in the access because the video the user sees on the television was deemed more important than the Z80's operation being delayed. Knowledgeable programmers would store time-sensitive code above the address 0x8000 to avoid the contention handling effects.[27](page 187)

The Z80's clock is obtained through the first bit of the horizontal counter, which results in a 3.5MHz synchronization signal. The ULA can detect accesses to the contented memory region by checking the CPU's control signals and address bus, in a circuit called the "contention handler". This technique was used by taking advantage of the Z80's timings. The Z80 prepares the address on the address bus half a clock before the memory control signals, allowing the ULA to stop it before memory is accessed.

Other techniques were considered to resolve this contention issue, but these wouldn't work to prevent access in time. The use of the WAIT input signal was one of these. However, it is only sampled by the Z80 a full T-state after the memory control signals are set, meaning the memory would have already been accessed by the time the CPU was told to wait. The BUSRQ signal was also considered to allow the ULA to have DMA and stop the contented access, but much like the WAIT signal, it gets sampled too late by the Z80.[27](page 188, 189)

I/O contention also exists, due to the ULA acting as a peripheral to the Z80. Whenever the ULA is accessing video data in memory, it cannot allow an I/O request from the CPU to take place at the same time, since it shares a data bus and control signals with the Z80 for the memory access. To mitigate this, the I/O request signal of the Z80 is also checked and considered for the contention handler circuit.[27](page 193)

There were 3 main versions of the ULA, named "Issues". Issue 1 had a bug with the data in ULA accesses by the Z80, such as keyboard reads, being erratic whenever it was read during contention when the ULA was updating the display. It wasn't initially observable since, normally, accesses to the ULA are made when the visible screen isn't being drawn, when the interrupt routine that increments the frame counter and scans the keyboard is triggered. As a hotfix, since discarding the first batch of ULA chips would be a heavy financial loss, a workaround was made on the circuit board to force the ULA to treat I/O requests as memory contention. The I/O request signal and the address bits A14 and A15 were tested (4000-7FFF address space) to force the ULA to

interpret this set of conditions as a memory contention and stop the CPU's clock. This modification, consisting of a 14-pin NAND gate integrated circuit placed upside down on the circuit board, was known as the "dead cockroach".
Issue 2 incorporated this modification internally. However, a new bug was discovered in both issues, related to the I/O contention. The specific ULA I/O port wasn't considered in the contention condition, so any other peripheral being accessed could be prevented from operating correctly. This was fixed by adding an OR gate soldered across the Z80's chip, known as the "spider" transistor modification [27](page 197-207).
Issue 3 had a redrawn interconnection layer since it used a more advanced ULA product from Ferranti. The ULA issue 3 also incorporated the "spider" mod in it, meaning that not all I/O accesses were contended. Some games used this feature, along with the floating bus behavior of the ULA, to read the contents of video RAM while the ULA was fetching the data, by reading a nonexistent I/O port, like 0xFF. By doing this, the screen updates could be synchronous to the display, rather than the interrupt, to avoid screen tearing and flickering graphics. These games were broken by later versions of the ZX Spectrum which would separate the busses using tri-state buffers (+2A and +3 models).[55]

2.4 Video

The screen has a working resolution of 256x192 pixels, in which color information is overlaid as a grid of 8x8 pixel regions known as attribute blocks. This working screen is comprised of 32x24 attribute blocks. In the screen memory region, visible in Figure 2.4, each byte represents a group of 8 pixels, corresponding to a line of an attribute block. The color memory region holds the colors of each attribute block. Only two colors per attribute block were possible, out of 8 possible colors. Each color is created with 3 bits, one bit per primary color (Red, Green, and Blue (RGB)). These two colors are called the *ink* (bits 2 to 0 in color data) and *paper* (bits 5 to 3 in color data) colors, where each 1 in a screen data byte corresponds to the *ink* color and each 0 to the *paper* color. Bit 6 in a color data byte determines if the colors used on the block are to be their 'bright' counterparts, making the possible number of colors 15, being that black has no bright variation. Bit number 7 of the color data of each attribute block determines if it is flashing the colors of the block. This switches the colors between *ink* and *paper* 3 times approximately every 2 seconds.[58]

The full active screen, without counting the border, is comprised of 49152 pixels. Screen memory can represent every pixel with 6144 bytes, since each byte holds information

for 8 pixels. The first 32 bytes of screen memory correspond to the top line of pixels of the screen with 256 pixels (0x4000 to 0x401F).

The next 32 bytes in memory (addresses 0x4020 to 0x403F) correspond to the top pixel line of the second row of attribute blocks, visually skipping 7 rows of pixels. This pattern repeats for 8 rows of attribute blocks. For simplicity, this group of 8 rows of attribute blocks will be referred to as an "attribute row group".

The second pixel-row of the first attribute block row is stored after the first pixel-row of the last attribute block row in the aforementioned "attribute row group" (8x32=256). Therefore, to draw the top pixel row (0x4000 for the first attribute block row) followed by the second in the same attribute block row, a 32-byte read on the address with 256 bytes of offset from the first one must be made (0x4100). So, in visual order, top to bottom, the screen data is stored in memory as such: 0x4000 to 0x401F, 0x4100 to 0x411F, ..., 0x4700 to 0x471F (end of first attribute block row), 0x4020 to 0x403F, and so on, until the last pixel row of the last attribute block row of an "attribute row group".

As aforementioned, the screen is composed of 32x24 attribute blocks. Therefore, the screen comprises 3 of these "attribute row groups" (3x8=24). These are stored in the areas 0x4000 to 0x47FF, 0x4800 to 0x4FFF, and 0x5000 to 0x57FF. The organization of rows of pixels is the same in each one of these groups.

Image Figure 2.6 illustrates the pixel lines of each attribute block row, their order, and their memory address offsets in relation to address 0x4000.

2.5 Keyboard

The keyboard of the ZX Spectrum is built into the computer's case. Different versions of the ZX Spectrum have different types of keyboards. The original model of ZX Spectrum 48k is in Figure 2.7, with rubber keys. The ZX Spectrum+ had a different casing, with a wider keyboard that fit more keys, allowing symbols that were only accessible with CAPS SHIFT and SYMBOL SHIFT as standalone ones, such as the arrow keys or the '.'. This keyboard's keys were made of plastic, instead of rubber, and can be seen in Figure 2.8.

ZX Spectrum keyboards possess one membrane consisting of a multiplexed matrix, where each row is pulled high and each column represents a half row. Each half-row of the keyboard is connected, through a diode, to a bit of the high byte of the address bus, while the rows are connected to a data bus feeding into the ULA.

The instruction set of the Z80 is defined for I/O ports to be designated by an 8-bit

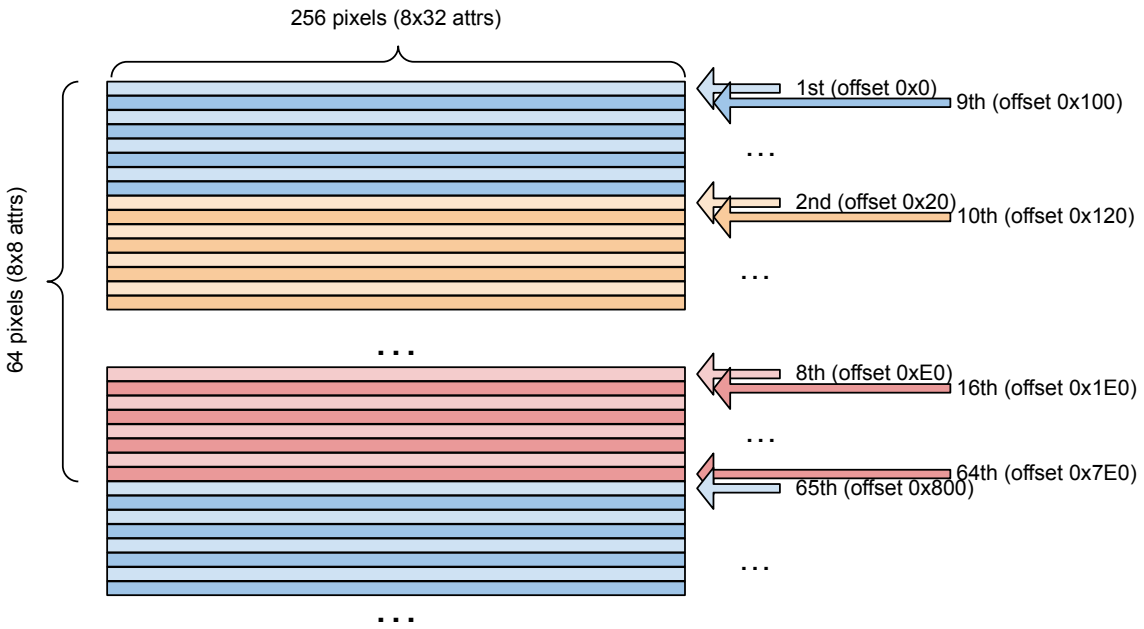

Figure 2.6: A representation of the lines of pixels on a screen with the order and offset in which their data is found

Figure 2.7: A ZX Spectrum 48k computer [9]

Figure 2.8: A ZX Spectrum+ computer [10]

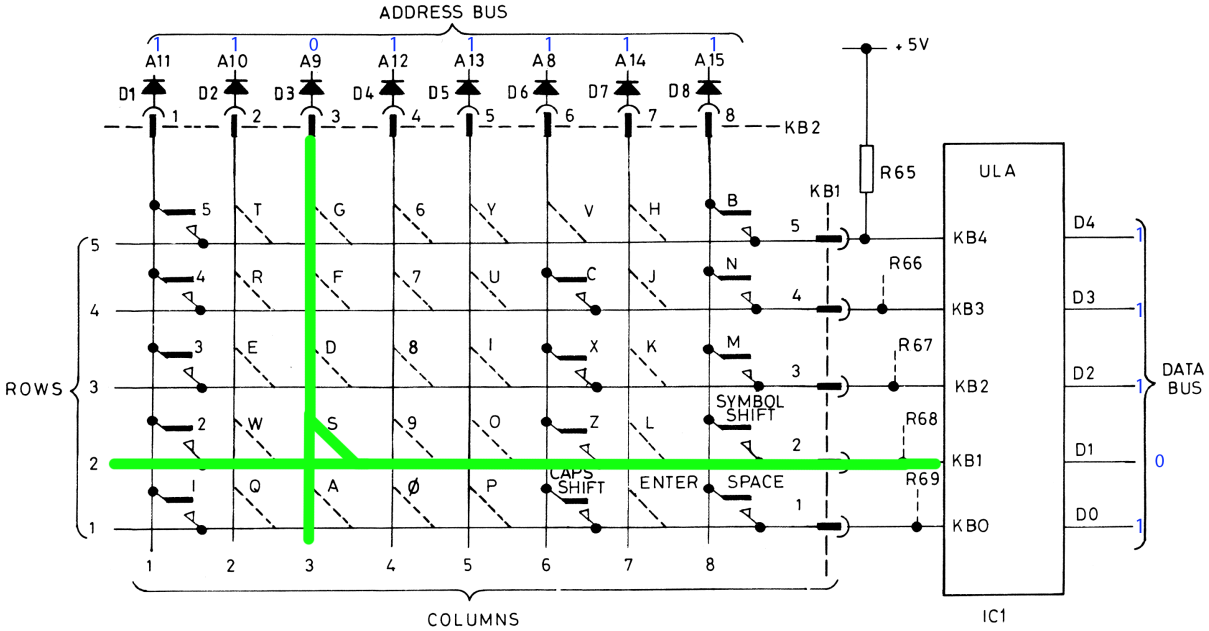

Figure 2.9: The ZX Spectrum keyboard's membrane schematic. Highlighted is the intersection between the half-row being read and the data bit for the S key [5]

number formed by the lower byte of the I/O address, despite the whole address bus being usable for addressing.[51] Therefore, the high byte was free to be used for keyboard half-row "selection". Figure 2.9 shows the row's connections to the ULA and the half-row's connections to the high bits of the address bus, as well as the pull-up resistors keeping the row's logic levels high, avoiding high impedance. It also shows the connection made after pressing the S key, explained in the next paragraph.

When a key is pressed and the address bit of the column to check is low, the column and row connect, pulling the value of that row low. Knowing the address that was set, the correct key can be determined based on what data bits are low. Reading the ULA's input port with different bit patterns in the high byte of the address can be done

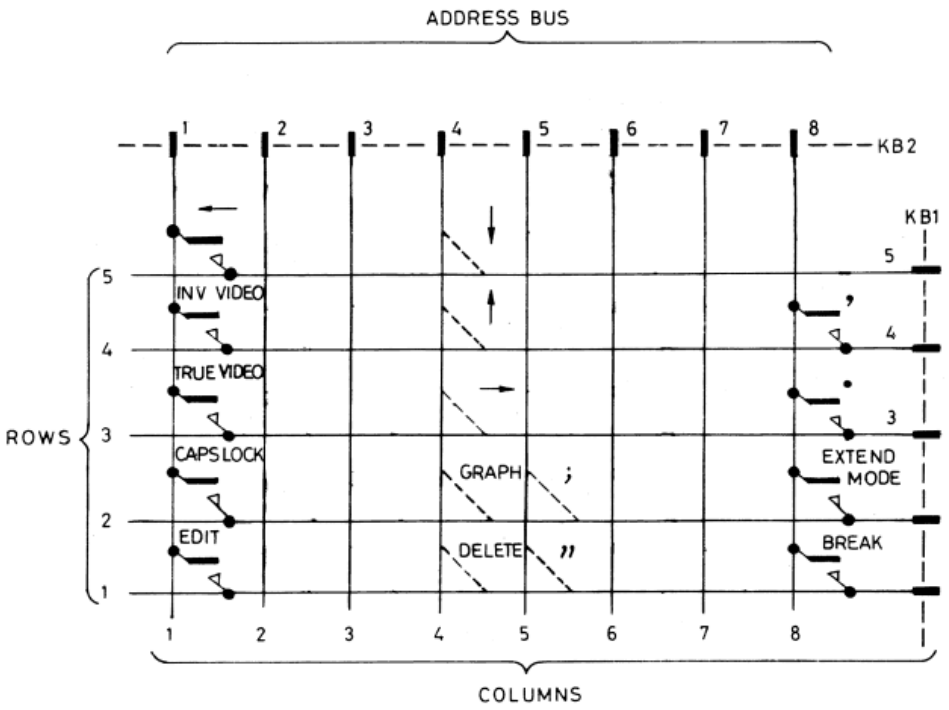

Figure 2.10: The ZX Spectrum+ keyboard's lower membrane[5]

to check if keys in multiple half-rows were being pressed, although it wouldn't be possible to distinguish each key press. The ULA makes this data available to the data ports when the CPU executes an IN operation at 0xXXFE. For instance, if the 'S' key is pressed, an IN operation at 0xFDFE will result in a data bus as follows: 11101. The address 0xFDFE corresponds to the A9 bit being low, "selecting" the half-row where S exists, as seen in Figure 2.11. Since D1 (data bit 1) corresponds to the 'S' key in the address/half-row being read, the CPU determines that the 'S' key was pressed.

The ZX Spectrum+'s keyboard changes were all possible on the keyboard alone, with no alteration required from the rest of the ZX Spectrum. This was done by having 2 different membranes: the top one being the same as the simpler ZX Spectrum keyboard and the bottom one forcing connections between normal keys and extended keys. For instance, the key for the up arrow (↑) in a ZX Spectrum+'s keyboard triggers the press of key 7 (row 4, column 4) and the key CAPS SHIFT (row 1, column 6). The connection between the up arrow and key 7 can be seen by comparing Figure 2.10 with Figure 2.9, as they both appear in row 4, column 4.

Both ZX Spectrum keyboard types suffered from phantom key presses, a phenomenon in which pressing a certain combination of keys on a multiplexed keyboard could lead to the shorting of another one that wasn't being pressed.

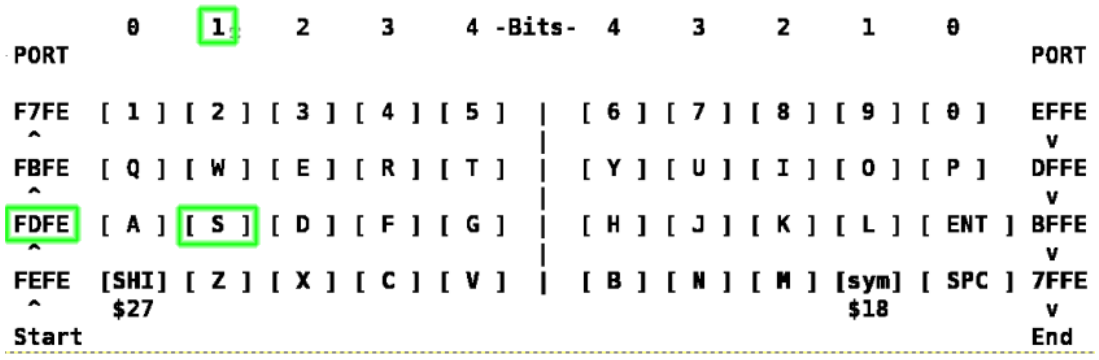

Figure 2.11: The ZX Spectrum keyboard with the addresses to check for each half-row[12]

2.6 Audio

The original ZX Spectrum emits sounds via a buzzer, capable of producing sounds up to 10 octaves (16.35Hz - 8372.018Hz).[47] The sound was generated by toggling the sound output signal at the frequency of the generated tone.[48] Since the ZX Spectrum models before the 128k didn't have a dedicated sound chip, the Z80 CPU had to process it. Although there was only 1 sound channel, others could be created through software routines, producing what would sound like more than one tone simultaneously. An example of this is a routine that was being worked on by Tim Follin in 1987 that allowed 6-channel sound with chorus bass, 128K snare drum, echo on/off/delay time, *portamento* and full ADSR (the envelope of the signal).[42]

Resistors connect the EAR and MIC sockets, so activating one activates the other.[4] The EAR input is connected to a 1-bit Digital-to-Analog Converter (DAC) inside the ULA to process the audio cassette's data. The resulting bit is made available to the CPU in bit 6 of the I/O port 0xFE, as seen in Table 2.1. The speaker would output sound when the signal to the MIC socket had enough amplitude to drive diodes D9 and D10 into conduction. This was done internally by executing an OUT instruction to the ULA's port (0xFE), with the data bit 4 low. These connections can be seen in Figure 2.12.

Toggling bit 4 using the OUT instruction produces sound. For instance, the note "middle C", which has the frequency 261.63 Hz, could be reproduced by toggling the beeper every $\frac{1}{523.26}th$ of a second ($523.26 = 261.63 \times 2$, duty-cycle of 50%). The Beeper subroutine, in the address 0x03B5 of the ROM, is responsible for playing a tone, given the values: Number of loop iterations (freq $\times$ duration 't' seconds) and the loop delay parameter (number of T states in the 'timing loop' divided by 4).[8]

Later, the 128k series ZX Spectrums were upgraded to have a dedicated audio chip, the

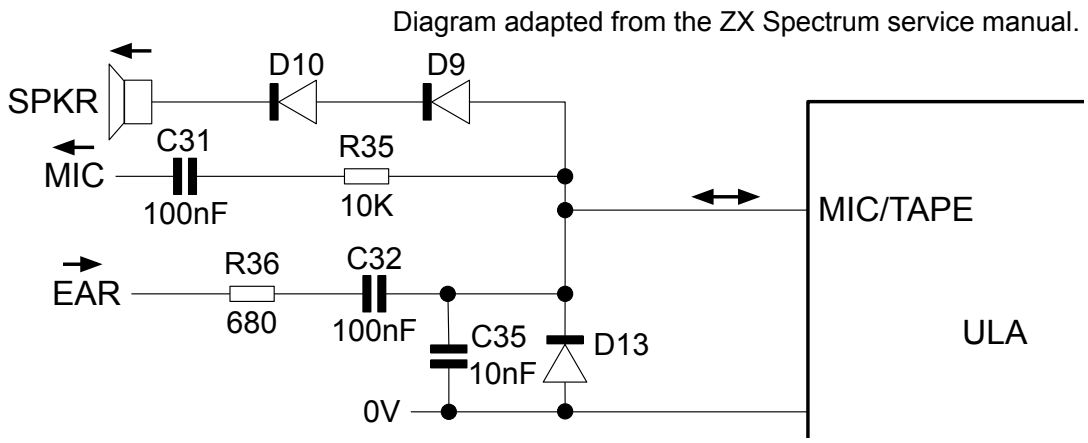

Figure 2.12: The connections between the audio tape input (EAR), the audio tape output (MIC), and the beeper to the ULA

Figure 2.13: The ZX Spectrum 128k menu

AY-3-8912.[59]

2.7 ZX Spectrum 128k's Main Menu

The ZX Spectrum 128k models have a main menu when they initialize. This menu included options for loading audio tapes, switching to the 128k's BASIC editor or the 48k's BASIC editor, using a calculator, and testing audio tapes. The menu can be seen in Figure 2.13.

ZX Spectrum 128k models had two pages of ROM. ROM 0 is new and contains the code used to generate the menu, among other things. ROM 1 is identical to the ZX Spectrum 48k's ROM, except for the previously unused area that started at address 0x386E. In this region, code was added for scanning and decoding of the keypad I/O expansion.

The disassembly of ROM 0 analyzed in this project was made by Matthew Wilson, with additional notes by Paul Farrow, available on his website.[1] In it, the main assembly routines can be identified:

- an "init" at address 0x2584 that sets up the system variables and the 128k's editor

flags. These flags, stored at 0xEC0D, contained information about the state of the menu: if the menu is displayed and if it is waiting for a key press;

- a "show menu" routine at address 0x259F that prepares the option text and option handler addresses, as well as the currently selected menu item's index. This index is stored at 0xEC0C and initialized as 0, representing the top menu option;
- a routine for "drawing the menu" at address 0x36A8 that receives an address to the menu's text and draws the menu's background, outline, the Sinclair stripes at the top, and text options;
- a routine at address 0x37CA that highlights the option specified by the first argument (register A), called by the previously mentioned routine as well as any action handler routine, for updating the selected option;
- a waiting loop routine, at address 0x2653, that waits for a key to be pressed by using the routine at address 0x367F. This waiting loop also produces the key click sound, retrieves its code and the menu keys action table, and executes the selected option's handler.

The options in a menu can be listed by using what was referred to as "text tables" in the studied disassembly. These resemble data arrays where the first byte represents the number of entries it has, and every string entry that follows ends with the hexadecimal code of the last letter of the string plus 0x80 as a terminator. The end marker of these types of tables is 0xA0, which is the space character ' ' with 0x80 added. An example of these text tables can be seen in Listing 2.1.

```
1  L2754:  DEFB $06          ; Number of entries.
2          DEFM "128     "   ; Menu title.
3          DEFB $FF
4  L275E:  DEFM "Tape Loade"
5          DEFB 'r'+$80
6  L2769:  DEFM "128 BASI"
7          DEFB 'C'+$80
8  L2772:  DEFM "Calculato"
9          DEFB 'r'+$80
10         DEFM "48 BASI"
11         DEFB 'C'+$80
12 L2784:  DEFM "Tape Teste"
13         DEFB 'r'+$80
14
15         DEFB ' '+$80      ; $A0. End marker.
```

Listing 2.1: The declared data variable for the main menu options' text

The tables that handle option selection and inputs also start with the number of entries. They contain pairs of identifiers and routine addresses. Since addresses have a fixed length, no terminator byte is required. Key action tables use key codes as identifiers, while option handler tables use the option's index. An example of this type of data can be seen in Listing 2.2. This data is used in the creation of the main menu shown in Figure 2.13.

```
L2744:  DEFB $05          ; Number of entries.
        DEFB $00
        DEFW L2831        ; Tape Loader option handler.
        DEFB $01
        DEFW L286C        ; 128 BASIC option handler.
        DEFB $02
        DEFW L2885        ; Calculator option handler.
        DEFB $03
        DEFW L1B47        ; 48 BASIC option handler.
        DEFB $04
        DEFW L2816        ; Tape Tester option handler.
```

Listing 2.2: The declatered data variable for the main menu options' handlers

2.8 Expansion Port

The ZX Spectrum computers had a socket designed for expansion peripherals. These expansions would connect to edge connectors on the back of the computer. Due to the way the ROM select bit was set on an expansion board, it could override the contents of the internal ROM and boot with another ROM. Sinclair Research produced two expansion boards named "ZX Interface I" and "ZX Interface II".

The ZX Interface I offered two network ports, allowing the daisy-chaining of other ZX Spectrum's for Local Area Network (LAN) communication. The network was called ZX Net. This interface also possessed a DE-9 RS-232 port and allowed up to 8 ZX Microdrives to be connected. ZX Microdrives were tape-loop cartridge drives created as a faster alternative to audio cassette tapes. An illustration of it can be seen in Figure 2.14.

The ZX Interface II (Figure 2.15) provided two joystick ports, as well as a ROM cartridge slot that allowed games to be loaded instantly, similar to the Sega Megadrive/-Genesis game console. The joystick ports were successful and many games advertised their support of 'Sinclair' type joysticks. Due to only 10 titles being released for ROM cartridges, and these being more expensive than the audio cassette tapes, the ROM cartridge slot was a failure.[3] The joystick ports did not adhere to the Kempston standard,

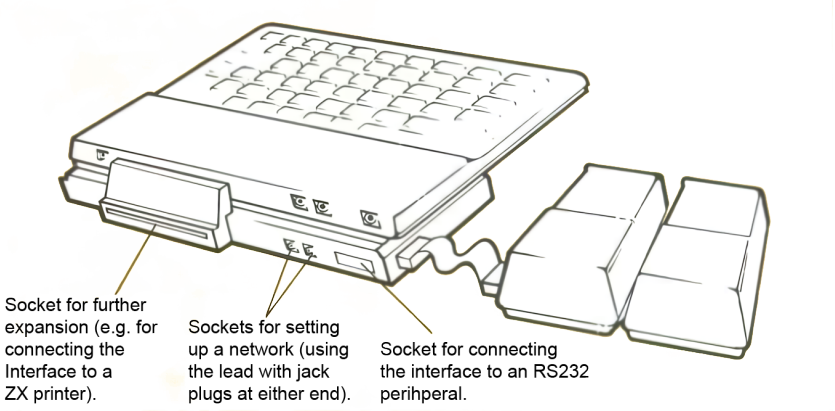

Figure 2.14: An illustration of a ZX Interface I connected to a ZX Spectrum[52]

Figure 2.15: The ZX Interface II[53]

	Joy 1	**Joy 2**
Left	6	1
Right	7	2
Down	8	3
Up	9	4
Fire	0	5

Table 2.2: Sinclair interface keyboard keys associations

Figure 2.16: The Kempston interface's circuit schematic[20]

thus making most joysticks at the time incompatible with this interface.[53] Instead of reading from an I/O port, Sinclair joysticks map directly to keyboard keys (0-9), as can be seen in Table 2.2, so games could implement them easily by reading the keyboard.

Kempston joysticks, pictured in Figure 2.16 used an interface that mapped to I/O port 0x1F. Software developers were required to read this port to obtain the state of the joysticks, unlike Sinclair joysticks. Despite this, due to the popularity of this interface and how easy it was to provide support for it, it was widely supported in most games.[3] A read from the port returns the state of the joystick's buttons (active-high). Depending on the specific Kempston interface, it could support up to 3 buttons, plus the 4 directional buttons for UP, DOWN, LEFT, and RIGHT, though typically only the FIRE button is supported. Figure 2.17 shows the schematics of this interface, as well as what the port data bits represent and each pin of the DB9 connector.

Printers, mass media storage, mice, and joystick interfaces for other joysticks were among the more popular types of expansion boards.

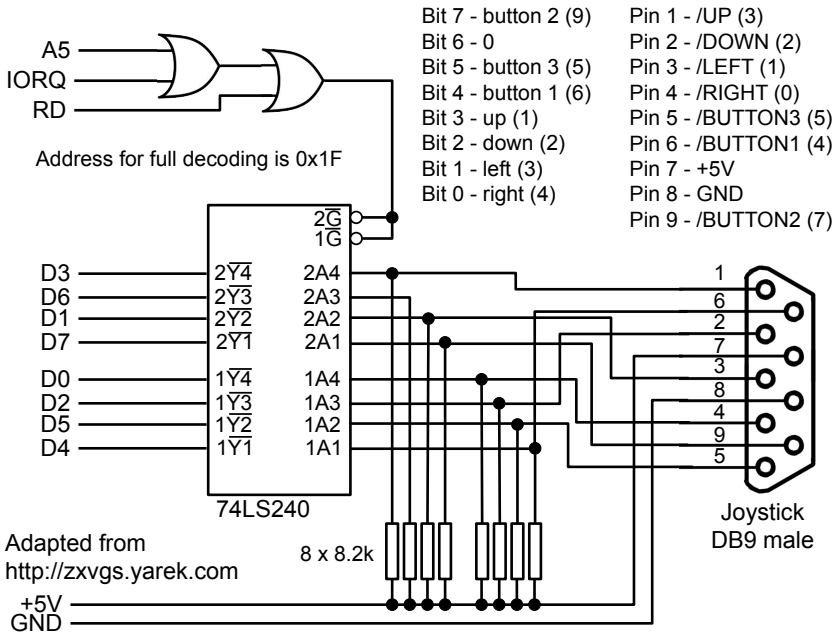

Figure 2.17: A Kempston joystick

2.9 Summary

This description of the ZX Spectrum illustrates the smart engineering involved in creating a computer at a time when technology was more expensive and rudimentary. Trade-off decisions that were made back then would no longer be required, but they make the system what it is today. It is important to keep the behaviors caused by these decisions in novel implementations of the ZX Spectrum for compatibility when executing its software. In the next chapter, other implementations are studied to determine their accuracy to the original.

3

Related Works

This chapter contains an analysis of different projects that attempted to recreate a ZX Spectrum, or components of it, on FPGA technology. This study was made to determine what open-source project is best as a reference for this implementation.

The implementations that follow in this section are all based on FPGA technology. Software implementations were not considered, based on the fact that they could not be used as a reference for this implementation. Software implementations also require an operating system to execute, which could lead to timing problems, as was previously mentioned. Only open-source implementations were studied in this analysis. The following subsections summarize, in general, the studied implementations.

3.1 ZX Spectrum 48K with ULAPlus by Andy Karpov

This[46] is an implementation of the ZX Spectrum 48k on a Cyclone IV FPGA, on an undisclosed board, with a 16-bit Video Graphics Array (VGA) (5-6-5 bits for RGB) output, a PS/2 connector, and an SD card reader, all written in VHSIC Hardware Description Language (VHDL). VHSIC stands for Very High Speed Integrated Circuit.

The CPU implementation used in this project is named "T80", a vendor-independent configurable Z80, written in VHDL, that supports the base instruction sets, undocumented instructions, and guarantees correct instruction timing. This CPU implementation is used in most of the projects analyzed here, which shows how tested it is. The implementation is available in open cores. [36]

Looking through the code, the CPU clock runs at 7MHz, whilst the ULA runs at 14MHz. However, this project lacked documentation of any kind, making it hard to understand and build upon. The creator, Andy Karpov, was contacted to obtain information on the project, but he quickly responded saying the project does not work as intended and that the GitHub repository it is available in was just a "sandbox" for him to experiment. For the reliability characteristic, the fact that the project does not work as intended will count as not being a full ZX Spectrum implementation. Despite this, it has great modularity and a great number of comments throughout the code, marking this project as having good maintainability and some usability.

3.2 Speccy2010 by Syd et al.

Speccy2010[29] is a ZX Spectrum clone created by Syd, a Ukranian developer. It is an improvement on one of his projects, the Speccy2007. The Speccy2010 is written in VHDL and uses a Cyclone II FPGA. In terms of peripherals, it has a VGA output of 24 bits (8 bits per color), two PS/2 ports, an SD card slot, and two 9-pin ports for joysticks. This project also uses the “T80” CPU implementation described in Andy Karpov’s project’s summary above.

The CPU runs at the correct clock speed of 3.5 MHz, having three additional turbo modes (7 MHz, 14 MHz, and 28 MHz). However, no information on the ULA speed was found in the project’s analysis, but a contention handler that was not accurate to how the ZX Spectrum works was found. In it, not only does the CPU “wait” for the ULA to finish, but the opposite also happens. The only code mentions of the ULA are in the top entity, seemingly used only for their contention handling, meaning no actual ULA module is implemented.

In terms of maintainability, most of the FPGA side of the project is modular, except for the sound-related components, which seem to all be used directly in the top entity. It gets a full point in modularity since the sound entities appear to be simple enough to not require separation from the top entity. Addressing the portability characteristic, besides the the Institute of Electrical and Electronics Engineers (IEEE) library, an Altera Mega Function is used in the code for the Phase-Locked Loop (PLL), which also should not be a problem given that the target platform is from Altera.

Based on the hardware features presented and a cursory analysis of the code, this implementation uses a microcontroller (STR755FV2T6) for some interaction with the SD card.

3.3 ZX Spectrum on FPGA by Mike Stirling

This project[34] was made by Mike Stirling on an Altera DE1 development kit. This board holds a Cyclone II FPGA, a 12-bit VGA output (4 bits per color), a PS/2 connector, and an SD card socket. SD card interaction is done through a recreation of the ZXMMC+ interface, an expansion board that can be plugged into a ZX Spectrum's edge connector to provide access to SD cards. The interfacing itself is done with ResiDOS, a ZX Spectrum BASIC extension that can be used to read TZX and TAP files, both of which are digital representations of audio tapes. ResiDOS must be installed as a tape the first time, but it installs itself to FLASH so it will always be there on future boots. ResiDOS only works on 48k mode in Mike Stirling's implementation.

Mike Stirling has a website where he briefly describes each component of his implementation, as well as issues the project has. No documentation besides this information has been found. Each part of this implementation is implemented modularly, despite most VHDL files being available in the root of the repository, showing a lack of file structure organization.

In this project, he fixes the contention that was present in the original hardware between the ULA and the CPU when accessing lower RAM. This was deemed necessary because of the RAM being fully present in a single chip, connected over a single bus. However, this causes problems with games that rely on this behavior.

There have been ports of this project to different boards: one for DE2-70 and one for DE2-115. The port for DE2-115, the same board as the target platform, was made by Stephen Eddy and it does not have any source code available, so it was not analyzed here. This does highlight the potential to use this project as a starting point, assuming Mike Striling's RAM access time slice would have to be altered for better accuracy. In terms of portability, this will gain a half-point on the target platform being used, since a port to it was created and some notes of that process were made on Stephen Eddy's blog.[33]

3.4 ZX-UNO by Superfo et al.

ZX-UNO[60] is an implementation of ZX Spectrum on a Xilinx Spartan-6 FPGA. It is a collection of implementations of old 8-bit systems that are organized as "cores". The ZX Spectrum core was written in Verilog, except for the CPU since this project used the aforementioned "T80" implementation. The implementation uses a 9-bit VGA (3

bits for each color) or composite video through RCA connector for video output, a PS/2 port for a mouse or keyboard, an SD card slot for loading software, and two 9-pin Atari joystick ports. It uses DIVMMC, another SD card storage interface like the previously mentioned ZXMMC+. There is documentation, but it only refers to the ZX-UNO as a whole and its hardware, with the only information about the ZX Spectrum core being the hardware peripherals supported for it.

The accuracy of the performance is sound since this implementation contains the contention handling, as well as the correct clock speeds for these modules. The organization of the repository has all versions of each core present in their folders, adding uncertainty to which sub-folder to use and which is the more recent and stable version. Inside there exists a folder named "common" which contains the main files for the implementation in its root, leading to a lack of modularity. In terms of portability, this project is less favorable due to using proprietary Xilinx libraries, incompatible with the target platform.

3.5 A-Z80 by Goran Devic

The A-Z80[6] is an implementation of the Z80 CPU on FPGA technology. The project contains an implementation of the ZX Spectrum as an example of the CPU's application. It is mostly written in Verilog and it targeted an Altera DE1 board. This implementation also supports a 9-pin interface for Kempston joysticks, but it does not support SD card reading. Loading software is done using the line-in port to play audio through audio cassette tapes or the Android app PlayZX which he developed for that purpose.

Through an analysis of the code, a note was found referring to the memory contention not being implemented yet, but the ULA and CPU clock speeds are accurate to the original, with a turbo option for the CPU also present which doubles that speed. The keyboard support from PS/2 to Spectrum takes into account keys available in US keyboards that correspond to combinations in the original Spectrum's layout, allowing quicker typing of characters such as "-", "_", "=", "+". Because of this, the user does not have to remember the original ZX Spectrum keyboard layout to write these symbols. This project has an overview, a user's guide, schematics, and other documentation making its usability seem good, but these documents are mostly for the CPU itself and not for the ZX Spectrum implementation.

3.6 ULA by Miguel Angel Rodriguez Jodar

This project[37] consists of an implementation of the ULA on a Xilinx Spartan-3 FPGA, using a Spartan-3 Starter Kit. This implementation was written in Verilog and was based on Chris Smith's "The ZX Spectrum ULA: How To Design A Microcomputer".[27] It is considered to be extremely accurate to the original.

This project's repository has many ZX Spectrum implementations. The 48k implementation has support for PS/2 keyboard, but none for VGA or SD Card, despite the spartan 3 starter kit having a VGA output. This project handles video (RGB) in the ULA module itself, in a very rudimentary way. Due to the uncertainty related to the status of the video output working or not after the analysis, the project is classified as "immature"/incomplete.

3.7 ZX Spectrum Next by Jim Bagley et al.

The ZX Spectrum Next[31] is a modern implementation of the ZX Spectrum for a Xilinx Artix-7 FPGA, written in VHDL. It uses an FPGA-based Z80 CPU with turbo modes up to 28 MHz, a proprietary operating system called NextZXOS, High-Definition Multimedia Interface (HDMI) or 12-bit VGA for video output, determined through cursory analysis of the code, a PS/2 port, an SD card socket, two 9-pin joystick ports, audio tape support, and an expansion slot for most ZX Spectrum peripherals. It has a case inspired by ZX Spectrum+.

This implementation supports not only tape files (.tap, .tzx) but also snapshot files (.sna, .z80).[32]

In terms of performance, the memory contention between the ULA and the CPU seems to be simulated appropriately and the clock speed of the CPU has four different modes, with the base one being 3.5MHz, accurate to the original ZX Spectrum. However, information on the clock speed of the ULA could not be found.

In terms of usability, this project has a lot of documentation, from the basics of programming to the peripherals and modules implemented, making it great for analysis comprehension. However, the code lacks comments compared to its density. The code is not commented enough to provide an understanding of the code on its own, which is why code commentary is rated as "Few". The analysis of the project's structure also suffers from the complexity of the code, making it harder to assess its modularity. Some components like keyboard-related, audio-related, and the HDMI video ones do

not follow a well-defined hierarchy. These are mostly implemented in the project's top entity, "zxnext.vhd", which contains 7128 lines of VHDL code. Modular architecture will therefore count as a 0 in the evaluation.

This project uses libraries specifically for Xilinx, mainly for the proprietary PLL and Block RAM (RAM) of their boards, leading to lower portability.

3.8 ReVerSe-U16 by Mvvproject

ReVerSe-U16[24] is a board focused on creating different consoles. The repository holds emulator implementations of the Atari 800, the MSX, the Nintendo Entertainment System (NES), and so on. It also contains implementations of ZX Spectrum models: a 48k with the "NextZ80" CPU implementation, and 48k and 128k implementations using the aforementioned "T80". Based on how commonly used the "T80" implementation is, the 48k implementation using it was chosen for this analysis.

The board itself hosts a Cyclone IV E as its FPGA, which is the same as the one present in the target platform. In terms of peripherals, it uses an HDMI port for video output, a Universal Serial Bus (USB) port for keyboard input, and a micro-SD card slot. Despite the video output being HDMI, the code converts from VGA in the top entity, with 6-bit color (2 per color), meaning it just as well supports VGA as it does HDMI. Some USB keyboards support the PS/2 interface on the same USB connector (Vcc and GND are used the same, D+ and D- that mapped to Clock and Data). The micro-SD card reader, despite being present in hardware, does not appear to be mentioned in the code of this implementation, so the method of loading the software is unknown.

The CPU, as described in the "readme" file of the implementation, runs at 50MHz, inaccurate concerning the original ZX Spectrum computers. No information on the ULA is present in the code, making it likely that I/O is handled directly between the CPU and the peripherals. Through cursory analysis, this does seem to be the case. The architecture of this implementation is modular and easy to follow, despite a lack of comments and documentation. The code is well-organized.

3.9 Comparison of the Related Works

A defined criterion based on ISO/IEC 25010:2011 system and software quality standard was used to classify the different implementations objectively.[2] The idea to use

this norm came from a dissertation by Duarte Alexandre de Sousa Galvão from Instituto Superior Técnico (IST) titled "ECSS-E-ST-50-15C Compliant CAN for Space Systems", where he used the norm to classify the library that was used as a base for the project, interpreting the norm in the context of his problem.

This norm is made for determining software quality originally, but the characteristics can be interpreted for hardware projects. The product quality characteristics were interpreted as follows:

- *Functional Suitability* - none of the projects analyzed have all the peripherals that this project aims to have. Some of the projects analyzed do not reconstruct a whole ZX Spectrum, with their focus being only certain components such as the CPU or the ULA. To avoid repeating others' work, a ZX Spectrum implementation with more peripherals and written in VHDL is more favorable.

- *Performance Efficiency* - due to the projects analyzed consisting of implementations of a retro computer emulator, performance and memory are not a direct issue. The performance mentioned in the norm refers to performance problems related to the amount of resources used under stated conditions, which does not fit this context. The sub-characteristic of "Time-behavior" was used to refer to the timings of the original ZX Spectrum, mainly the clock speed of the ULA, the CPU, and their contention in lower RAM. These timings were determined through cursory analysis of the code of each project.

- *Compatibility* - this characteristic was interpreted to relate to the ZX Spectrum implementation's specification. Projects that implement the ZX Spectrum model needed, specifically 48k, are favorable.

- *Usability* - this is related to the accessibility, learnability, and operability of the Projects. Documentation, written information, and comments are considered, assuming these are focused on the ZX Spectrum implementation specifically, since some projects could focus mainly on multiple computers and consoles.

- *Reliability* - the maturity sub-characteristic is used to qualify the "completeness" of the projects and therefore how reliable they are. The projects might not be focused on ZX Spectrum implementations. In this context, only completed projects were considered. Other sub-characteristics such as availability, fault tolerance, and recoverability are not adequate for this analysis due to not applying to the projects.

- *Security* - this characteristic is irrelevant since this project is to be deployed in a local environment (FPGAs) and there is nothing vital to protect.

- *Maintainability* - an analysis of each project's entities and components was made to determine their modularity. Other sub-characteristics of the norm ISO/IEC 25010:2011 [2] such as reusability, analysability, and modifiability are a must-have for all projects to be considered in this study.

- *Portability* - is related to the hardware specifications used in the project. The manufacturer of the target platform used (Xilinx or Altera) and the usage of proprietary Intellectual Property (IP) (Altera/Intel) Blocks, such as PLL, BRAM and Digital Signal Processors (DSP) are considered. IP blocks from Altera are preferable for facilitating the project's use on the target platform. Moreover, this characteristic also accounts for the specific details of each peripheral, such as the VGA DAC's input encoding. This characteristic is quantified as a percentage of the number of peripherals that match the target platform's specifications over the total number of peripherals it implements. Projects that were implemented for the target platform are preferred. This characteristic is not to be confused with functional suitability, for this is more hardware-related as opposed to the implementation of the interactions with each peripheral.

The evaluation of each project following these characteristics can be found in Table 3.1. Table 3.2 contains a legend for the symbols used in the comparison table. For the usability characteristic, "None", "Weak", "Few", and "Good" were used to rate the features. The legend for the Project IDs can be seen in Table **??**.

The scoring was done on a scale from 0 to 5. Each feature was evaluated from 0 to 1, and the full characteristic scores were calculated by averaging these features and multiplying them by 5. Each '✓' counts as a full point, '~' as a half point, and '?' and 'x' count as no point (it could be said that the '?' is pessimistic). In terms of the usability characteristic, "Good" counts as a full point, "Weak" and "Few" count as half a point, and "None" counts as zero points. The final scores were the result of averaging each characteristic's score and are displayed in Table 3.3. The legend for the Project IDs is in Table **??**.

The project "ZX Spectrum on FPGA by Mike Stirling", got the highest score. Therefore, it has been chosen as a reference for this one.

Table 3.1: Comparison between the different projects

	Projects							
Feature	**ZX Spectrum 48k with ULAPlus by Andy Karpov**	**Speccy2010**	**ZX Spectrum on FPGA by Mike Stirling**	**ZX-UNO**	**A-Z80 by Goran Devic**	**ULA by Miguel Angel Rodriguez Jodar**	**ZX Spectrum Next**	**ReVerSe-U16**
Functional Suitability								
Written in VHDL	✓	✓	✓	x	x	x	✓	✓
VGA	✓	✓	✓	✓	✓	x	✓	✓
PS/2	✓	✓	✓	✓	✓	✓	✓	✓
SD Card	✓	✓	✓	✓	x	x	✓	x
Performance Efficiency								
CPU Clock @ 3.5MHz	x	✓	x	✓	✓	✓	✓	x
ULA Clock @ 14MHz	✓	?	✓	✓	✓	✓	✓	x
Memory Contention	✓	x	x	✓	x	✓	✓	x
Compatibility								
Spectrum 48k	✓	✓	✓	✓	✓	✓	✓	✓
ULA	✓	x	✓	✓	✓	✓	✓	x
Z80	✓	✓	✓	✓	✓	✓	✓	✓
Usability								
Documentation	None	None?	None	Weak	Weak	None	Good	None
Information	None	Weak	Good	Weak	Good	Weak	Good	Weak
Code Commentary	Good	Few	Good	Few	Good	Good	Few	Few
Reliability								
Full ZX Spectrum	x	✓	✓	✓	✓	x	✓	✓
Maintainability								
Modular Architecture	✓	✓	✓	✓	✓	✓	x	✓
Portability								
Compatible Specifications	66,7%	100,0%	66,7%	66,7%	66,7%	33,3%	66,7%	33,3%
Target Platform used	x	x	~	x	x	x	x	x
IP PLL for Altera used	✓	✓	✓	x	✓	✓	x	✓
IP BRAM for Altera used	✓	✓	✓	x	✓	x	x	✓
IP DSP for Altera used	✓	✓	✓	x	✓	✓	?	✓

Symbols	
✓	Complete or nearly complete feature fulfillment.
x	Feature lacking.
?	Unknown due to not being found in analysis.
~	Feature has been fulfilled but is not present in the implementation.

Table 3.2: Legend for the above table's symbols

Table 3.3: Simplified projects evaluation, scored 0 to 5

	Projects							
Characteristics	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Functional Suitability	5,00	5,00	5,00	3,75	2,50	1,25	5,00	3,75
Performance Efficiency	3,33	1,67	1,67	5,00	3,33	5,00	5,00	0,00
Compatibility	5,00	3,33	5,00	5,00	5,00	5,00	5,00	3,33
Usability	1,67	1,67	3,33	2,50	4,17	2,50	4,17	1,67
Reliability	0,00	5,00	5,00	5,00	5,00	0,00	5,00	5,00
Maintainability	5,00	5,00	5,00	5,00	5,00	5,00	0,00	5,00
Portability	3,06	3,33	3,47	0,56	3,06	1,94	0,56	2,78
Average Score	**3,29**	**3,57**	**4,07**	**3,83**	**4,01**	**2,96**	**3,53**	**3,08**

3.10 Summary

This chapter presented and compared several attempts to recreate the ZX Spectrum. It focused only on the most complete ones. Analyzing all the projects gave an idea on different solution directions and new ideas to explore. An outcome of this study was the selection of the T80 CPU implementation on FPGA.[36] The next chapter describes the target platform in further detail, contextualizing the evaluation further.

4

The Target Platform

The target platform used in this project is from Terasic and is designated DE2-115, as shown in Figure 4.1. This board was chosen due to its peripherals, avoiding the need to add extra circuitry. The Terasic board DE1 was considered because it is smaller than the DE2-115 so it could fit inside the case of a ZX Spectrum. However, due to its lack of on-board peripherals, it would be necessary to create hardware for it.

The onboard peripherals used for this project are the VGA output for video, the PS/2 port for keyboard input, the SD card reader to load software, and the line in/out for audio I/O.

The FPGA programming was done through Intel Quartus. It is a design software that supports many different FPGA chips from Intel and performs synthesis of Hardware Description Language (HDL). It supports VHDL and Verilog. Quartus allows the instantiation of NIOS II embedded processors in Cyclone IV devices by using its Platform Designer. Software can be developed for the processor by using a built-in version of the IDE Eclipse, which also allows live debugging.

4.1 FPGA

An FPGA is an integrated circuit that can reconfigure the hardware inside to create a custom logic system. In 1984, Altera introduced Programmable ROM (PROM) and Programmable Logic Device (PLD)s. They are referred to as 'field programmable' for

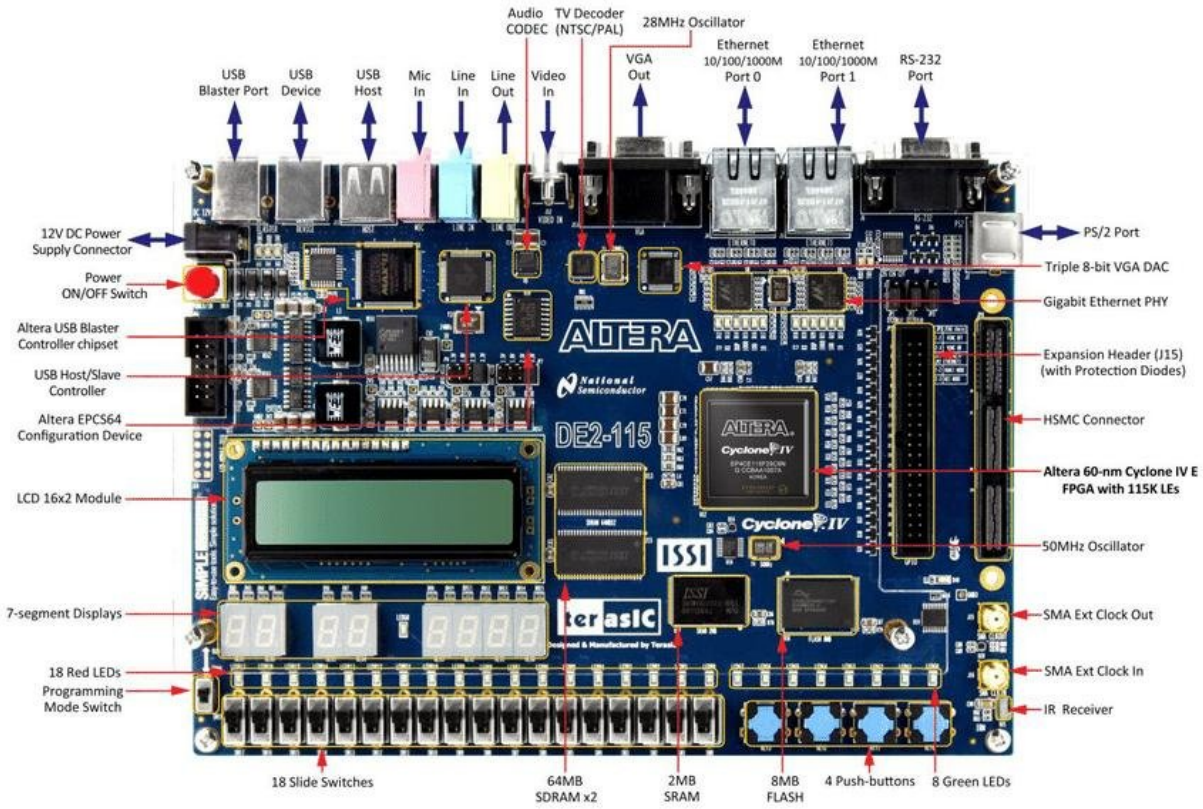

Figure 4.1: Altera DE2-115 and it's components[13]

being programmable "on the field", allowing the user to configure it to perform anything from simple logic gates to complex circuits on site. The technology is often used in hardware prototyping, hardware acceleration, and updating designs in remote deployment locations, including satellites.[7]

An FPGA chip contains multiple Configurable Logic Block (CLB)s, which in turn contain many Logic Element (LE)s, memory blocks, and DSP blocks that are all connected through a configurable mesh of interconnections. A simplified illustration is in Figure 4.2. The CLBs are the smallest most prominent and fundamental blocks in an FPGA chip. They are configurable primitive cells, used to produce elementary computation.

4.2 Intel Cyclone IV

The target board contains a Cyclone IV EP4CE115 FPGA chip from Intel. These devices contain 114,448 LEs, 3,888 Kbs of embedded memory, 4 general-purpose PLLs, and 528 user I/Os. Its LEs each contain four-input Look-up Tables (LUTs) which can implement any logic function of four variables, as well as a programmable register and the aforementioned connections with other LEs. The structure of these LEs is pictured

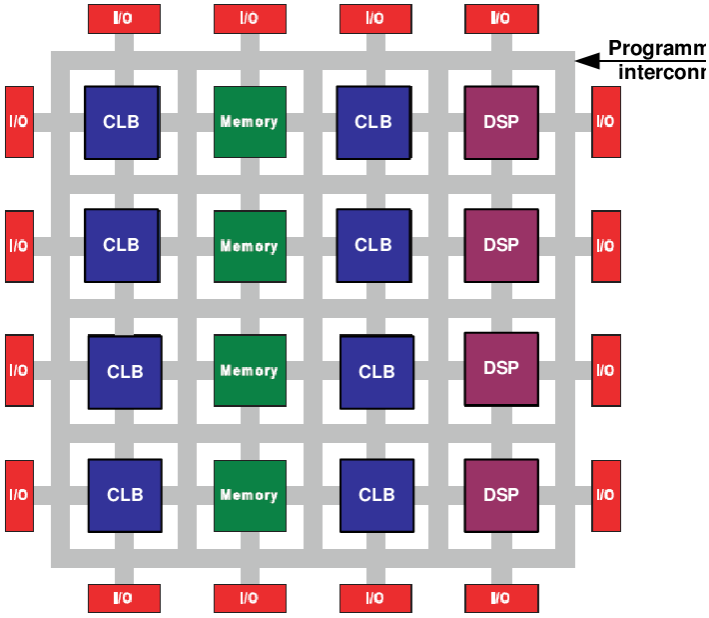

Figure 4.2: The general structure of an FPGA chip [16]

in Figure 4.3.

4.3 VGA output

VGA video is formed by scanned frames, which are composed of lines, each line being a sequence of pixels. The lines are rendered from top to bottom and the pixels from left to right. What determines a change of line or a change of frame are two separate sync pulses: hsync and vsync, respectively.

Initially, to render anything on a VGA monitor, the proper timings are necessary for synchronization. These timings can be counted in pixels, considering the pixel frequency of a chosen resolution. The resolution 1024x768 at 60 Hz refresh rate, has a pixel clock frequency of 65 MHz. The timings consist of four types: Active Video, Front Porch, Sync Pulse, and Back Porch. The active video takes place whilst the visible pixels are being rendered on screen. The front porch and back porch are periods where blank pixels are transmitted, but not shown, known as the blanking region before and after the sync pulse hsync and vsync. The screen regions associated with these timings are illustrated in Figure 4.4.

The VGA Digital-to-Analog Converter (DAC) in this platform allows 24-bit color, 8 for each primary color, Red, Green, and Blue (RGB). The VGA signals go from the FPGA to a high-speed video DAC. The VGA vertical and horizontal sync signals are sent

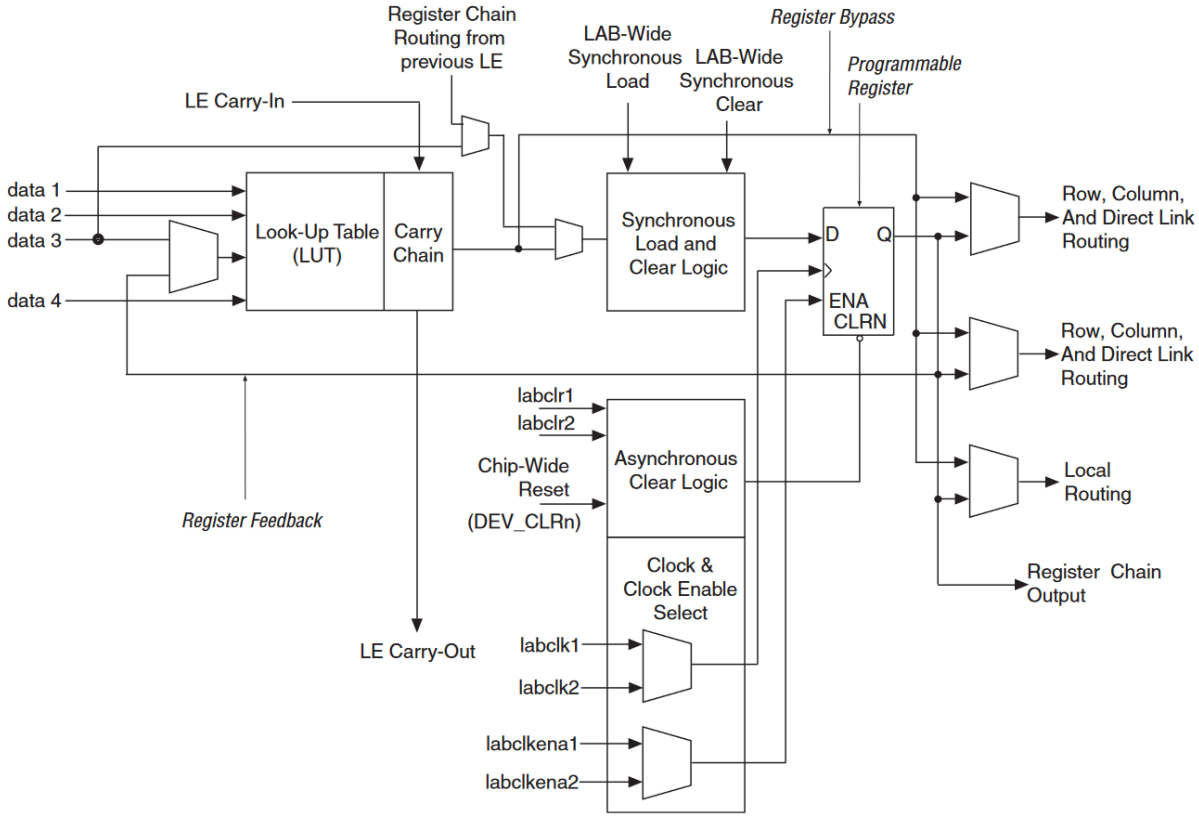

Figure 4.3: LEs in Cyclone IV devices [11].

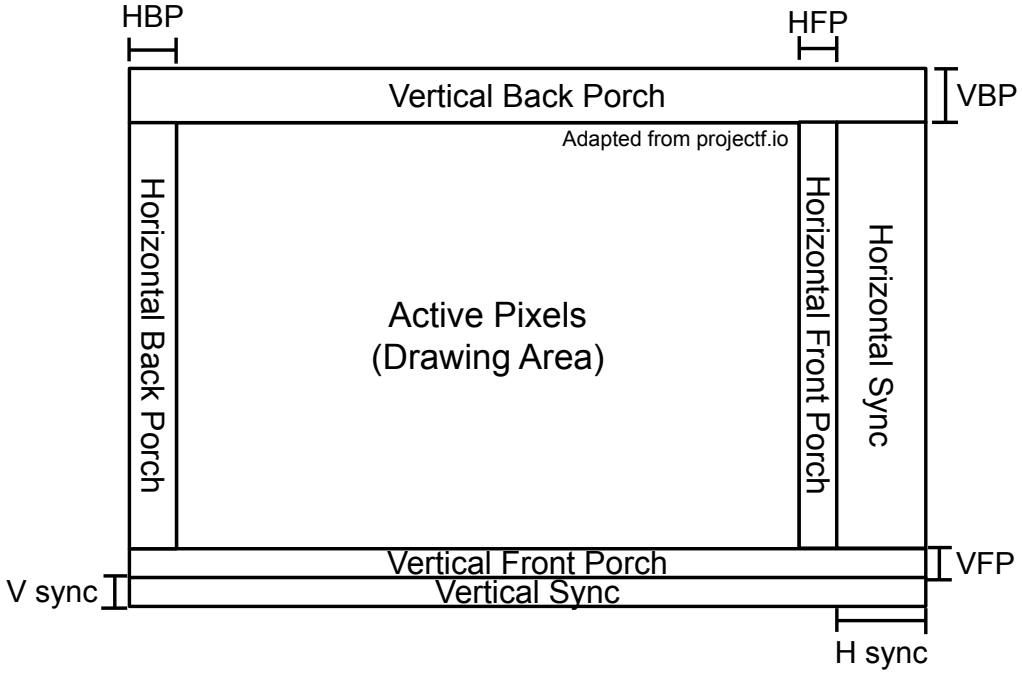

Figure 4.4: The VGA timings in the context of a full screen[17]

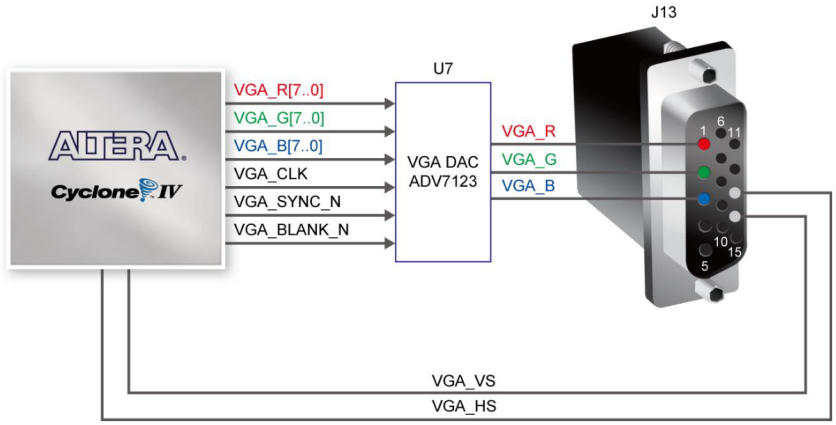

Figure 4.5: Connections between FPGA and VGA[13]

directly to the VGA port, as shown in Figure 4.5. This DAC supports up to 1280*1024 resolution, the SXGA standard.

4.4 Audio I/O

This board is equipped with an audio controller (Wolfson WM8731) which is capable of doing both Analog-to-Digital Converter (ADC) and DAC and supports high-quality 24-bit audio. It supports an adjustable sample rate from 8 kHz to 96 kHz.[41]

It is controlled directly by the FPGA via serial Inter-Integrated Circuit (I2C) bus interface. These connections are displayed in Figure 4.6. It includes a mute function as well as adjustable line-level volume control. The XTI/MCLK input is set up according to the desired sample rates of the internal DAC and ADC.

The WM8731 offers 4 digital audio interface modes that are configurable through software: Right Justified, Left Justified, I^2S, and DSP mode. This board lacks connections to the configuration interface of the chip, so it can only be used in its default state. The digital audio interface takes data from the internal ADC digital filter, placing it on the ADCDAT output. This output contains the formatted digital audio stream with left and right channels multiplexed together. The same happens with the internal DAC and DACDAT signal, in the opposite direction (input). DACLRC and ADCLRC are alignment clocks that control whether Left or Right channel data is present on either DACDAT or ADCDAT. The data and stereo clocks are synchronized with each other with the BCLK (bit clock). BCLK and the LR clocks can be either inputs or outputs depending on whether the device was configured to be in slave (peripheral) or master (controller) mode. By default, it is in peripheral mode.

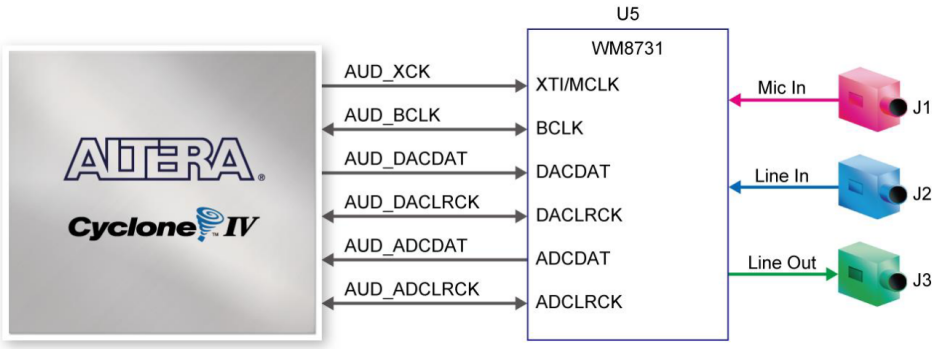

Figure 4.6: Connections between the FPGA and the Audio CODEC[13]

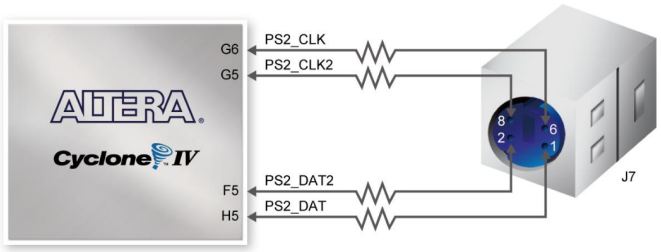

Figure 4.7: Connections between FPGA and PS/2 [13]

4.5 PS/2 Keyboard/Mouse Port

PS/2 is a serial synchronous communication norm used for transferring data between a computer and a keyboard or a mouse. PS/2 port signals consist of a clock and data, for the connected device. If a splitter is used, both peripherals can interact with a host through one port.

In computers, when a key is pressed on a PS/2 keyboard, a code is sent to the host CPU. If the key is held down, its code will be continuously sent to the host according to a defined auto-repeat rate. When a key is released, a code is sent to the CPU, known as a 'break' code, to inform of a key's release. These codes consist of the key's base codes with 0xF0 as a prefix. For extended keys, like SHIFT and CTRL, their codes have a prefix of 0xE0, and their 'break' codes have a prefix of 0xE0F0 [23].

The PS/2 protocol is half duplex, meaning the host can also send commands to a keyboard, such as reset, disable, enable, set scan code set (there are 3 different key scan code sets), and set LED states (Numlock, Capslock, and Scroll lock) to name a few.

The DE2-115 board includes a standard PS/2 interface and supports, through a single PS/2 port, up to two devices at a time, most often a mouse and a keyboard. The connections from the FPGA to the PS/2 port are shown in Figure 4.7.

	Full SD	miniSD	microSD	SD Card Capacity
SD				up to 2GB
SDHC				High Capacity 4GB to 32GB
SDXC				Extended Capacity Over 32GB up to 2TB

Figure 4.8: Different SD card types[25]

4.6 SD Card Socket

An SD card is a proprietary flash memory card format developed by the SD Association (SDA). The standard was introduced in August 1999 (an improvement over MultiMediaCards (MMCs)) and has become the industry standard. They come in different physical size formats, as well as storage sizes and speeds, as illustrated in Figure 4.8.

The SD card protocol consists of an exchange of tokens between the card and the host. These tokens can either be command tokens or response tokens. Data transfers occur in packets composed of a data block and Cyclic Redundancy Check (CRC) bits. There is a transfer mode that allows multiple blocks of data to be sent at once, requiring the host to send a stop command at the end. Data is transferred through DAT pins, of which there are 4 (bit width configurable), whilst commands and responses are transmitted through the CMD pin. These transfers are done with the most significant bit first[26]. SD cards can also communicate in Serial Peripheral Interface (SPI) mode by interpreting its I/O pins as SPI signals. In this mode, DAT3 is used as a Chip Select (CS), DAT0 as a data output pin, and CMD as a data input pin.

The target platform includes an SD card holder needed for interacting with full SD cards, which is displayed in Figure 4.9. The system must implement a custom controller that accesses the SD card either in SPI mode, 1-bit mode, or 4-bit mode. The connections with the FPGA are shown in Figure 4.10. The NIOS II is a small embedded processor that can be instantiated on the FPGA. This CPU can be used as a controller on the system, for example, to automate the access to the SD card by using libraries.

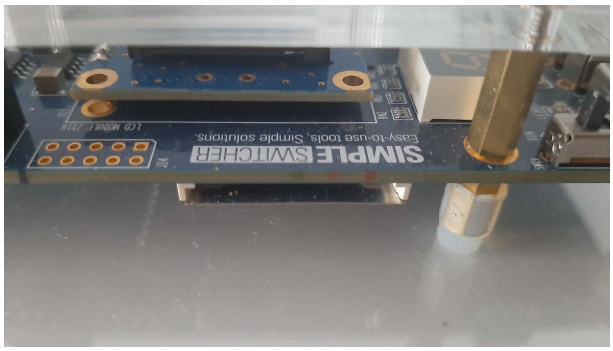

Figure 4.9: The SD card slot below the target platform, as seen from the left side of the board

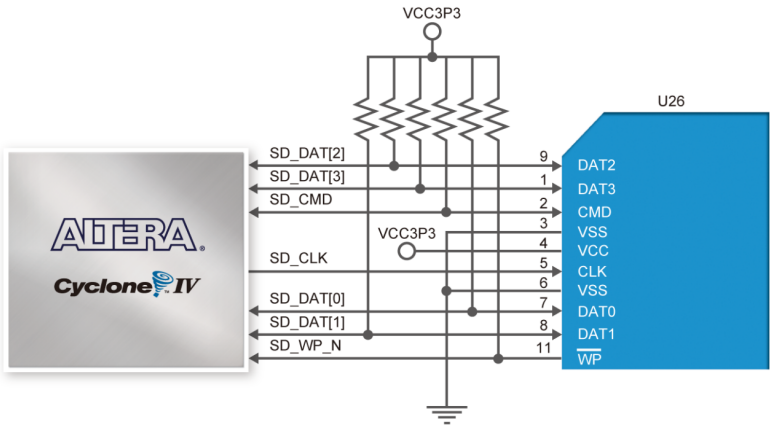

Figure 4.10: Connections between FPGA and SD Card Socket[13]

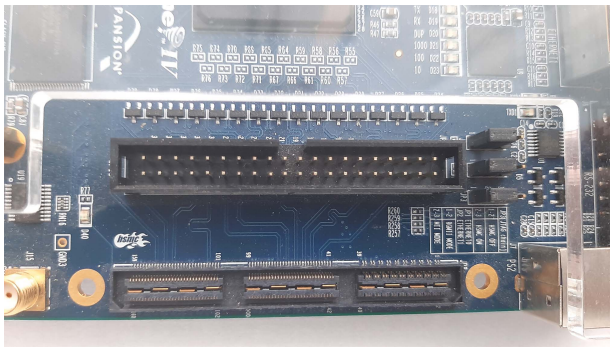

Figure 4.11: The expansion header of the target platform

4.7 I/O Expansion Header

The board possesses a 40-pin expansion port, observable in Figure 4.11, with configurable I/O standards. This port is used to connect a PCB with ports to connect two joysticks and a native ZX Spectrum keyboard.

4.8 Summary

The history and development of PLDs has come a long way, from ULAs which could only be programmed once, to Programmable Array Logic (PAL) which had variants that allowed re-use, to Generic Array Logic (GAL) that were re-programmable, to FPGAs that have a great capacity and number of LEs with re-programmable connections between them for easy prototyping and design changes.

5

Proposed Modernized ZX Spectrum Fusion Architecture

The purpose of this chapter is to explain the architecture proposed for the realization of a modernized ZX Spectrum and how it is structured. This chapter starts with a "high-level" overview of the architecture and then details each of its aspects. This chapter also serves to explain what parts of the selected base project were reutilized for this one.

5.1 Architecture Overview

The conception of the architecture was defined around the peripherals that were present on the target platform (shown in Chapter 4). Peripherals such as a basic input (keyboard) or basic video output, were considered more important to implement first for being required to operate and "interface" with a computer. The main focus of implementing these novel peripherals is how to "transform" what the ZX Spectrum natively uses to interface with them. The ULA in this project maintains the internal interactions with the Z80 for accuracy. However, it is not a replica of the original one due to the new video output requiring faster clock speeds and a different signal generation interface. The outputs of the ULA must be processed for the peripherals by other components.

In Figure 5.1 there is a simplified schematic of the full architecture. The rectangles localized outside the DE2-115 board's rectangle (in blue) represent the peripherals of the

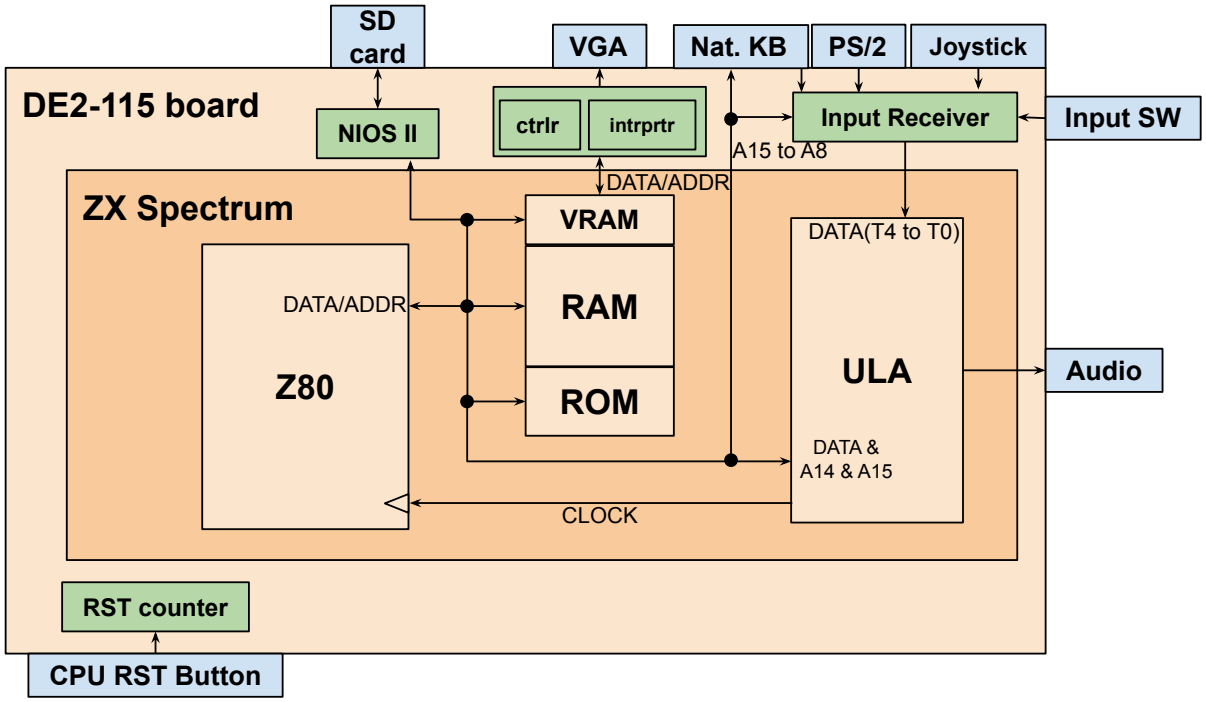

Figure 5.1: A simplified overview of the proposed architecture of this project

target platform and the rectangles outside the ZX Spectrum area (in green) represent modules that do not belong to the ZX Spectrum itself.

The arrows in this image are a simplified representation of connections between the modules. Extra details, like the size of the busses or separate lines for connections between the same modules (ADDR/DATA) are omitted for the sake of simplifying the diagram.

The memory components accessed by the Z80 and the ULA are implemented using a dual port to allow simultaneous accesses from both components, simplifying the interactions. The memory contention present in the ZX Spectrum was a consequence of the system architecture and the technology available at that time. To preserve accuracy, the contention is replicated with the same conditions the original ZX Spectrum had.

5.2 CPU

The CPU has the same signals as the Z80 chip present inside the original ZX Spectrums. In terms of the processor itself, the implementation that is being used is the "T80", the vendor-independent configurable Z80 implementation, written in VHDL, used in most of the analyzed projects.[36]

5.3 RAM

The ZX Spectrum's RAM is implemented with BRAMs on the FPGA. The BRAMs support dual-port access, which allows simultaneous access to the memory contents via two independent busses. This is convenient to provide access to the shared video memory. The video and color memory regions (6,144 bytes and 768 bytes, respectively) are instantiated as separate BRAMs to allow the video IP Block to fetch data from both simultaneously, in time to render it to VGA. This was deemed important due to the high pixel clock frequency of the novel peripheral. The remaining RAM is implemented as a single BRAM with 42,240 (0xA500) bytes.

5.4 Video Output

The video output is sent from the video component with all the signals necessary for the DAC and the port, as detailed in Section 4.3. A resolution of 1024x768 was selected not only for being a common resolution of VGA monitors and being available in the DAC of the target platform but also because this screen resolution is exactly 16 times larger than the ZX Spectrum's resolution, as observable in Figure 5.2. A screen area of 16 times the original corresponds to an area where each dimension was multiplied by 4. Each ZX Spectrum pixel corresponds to a 4x4 square of pixels on the screen.

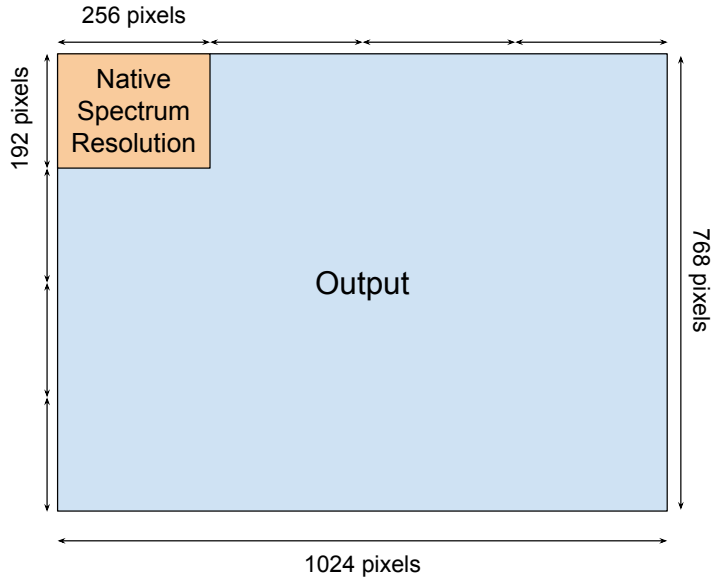

Figure 5.2: The ZX Spectrum resolution compared to the proposed one

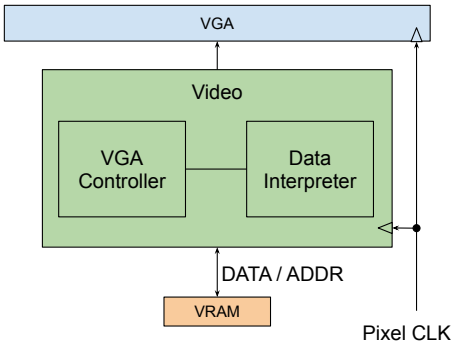

Figure 5.3: A simple diagram showing what the video module contains and its connections with other components

Note that the "Native Spectrum Resolution" area shown in the image only considers the active screen area and not the border the ZX Spectrum has. The border, in this project, was determined to be unnecessary. A 1-pixel thin border could be added around the image to maintain the resolution of the active video output, as an option the user could toggle. Since every ZX Spectrum pixel is 4x4, 1 pixel of border does not affect screen visibility. Other screen sizes can be added easily as long as they are powers of 2 in relation to the original resolution. This would allow a larger border to be output for users who prefer it.

For the data to be sent to the VGA DAC in the established resolution, a pixel frequency of 65 MHz is required. This means that a module separate from the ULA is required to handle the video rendering and data fetching since the ULA's internal clock frequency of 7 MHz would be too small. Therefore, the screen data in RAM is no longer read by the ULA. This module is pictured in Figure 5.1 as the rectangle containing a controller and an interpreter (on top, in green). The contention control is still performed by the ULA regardless to match these accesses with the original ZX Spectrum. The ULA is also responsible for other operations like the sound I/O. A simplified model of the video component and its connections can be seen in Figure 5.3.

5.5 Keyboard and Joystick

The keyboard input on the proposed architecture is done with either a PS/2 keyboard or a native ZX Spectrum. For the latter, simple connections with the address bus and the ULA are sufficient, through the use of a small expansion board for the target platform to plug the membranes in.

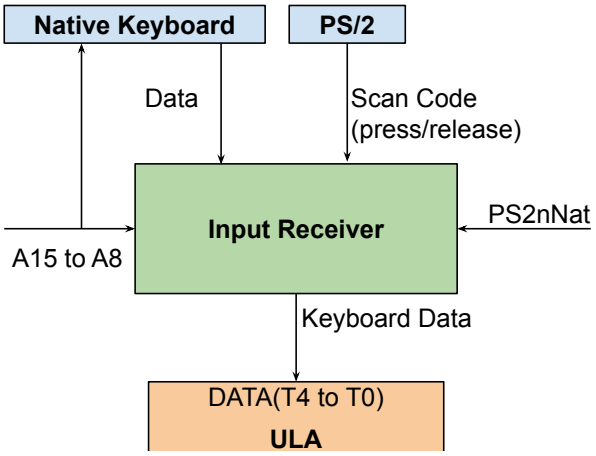

Figure 5.4: A simple diagram showing what the PS/2 converter module should receive and produce

In terms of the PS/2 keyboard, the key codes are interpreted into the correct data values for the ULA to receive, depending on the address bus. Therefore, an IP core was created to receive the PS/2 key codes and the high byte of the address bus, and send the translated keyboard data output to the ULA. A diagram of what this module should receive and output, knowing that the user can toggle between a PS/2 keyboard and a native keyboard for input, can be seen in Figure 5.4.

It was decided that the keyboard module should manipulate the LEDs on the keyboard, toggling the lights for Numlock and Capslock states. This assists the user's interaction with the keyboard, mainly the Numlock status, for convenient Number Pad usage. This requires two-way communication between the converter and the keyboard itself, through means of a PS/2 Controller that allows this.

A Sinclair joystick works similarly. A joystick connected through an expansion board allows its inputs to pass through the keyboard module and trigger the corresponding half-row's data bits according to the buttons pressed. Section 2.8 has the keyboard keys Sinclair joysticks map to. The conversion of the joystick's state is handled by the "Input Receiver" in Figure 5.1.

To interpret joysticks as Kempston interface joysticks, the inputs are converted to the Kempston format and exposed on hardware port 0x1F, as discussed in Section 2.8, so the Z80 may read them. This is handled by a separate component for the Kempston Interface itself, since it is separate from the ULA and, by extension, the keyboard.

5.6 SD Card

To read ZX Spectrum program files from SD cards and load them, it is necessary to have hardware and software support that was not on the original ZX Spectrum 48k. Hardware refers to the electrical level control of the SD card, whereas the software refers to the programming of the file systems on the SD card, as well as an interface through the Z80 processor to load the files themselves. The software to access the files is executed on a soft processor on the FPGA, the NIOS II. This processor assists in loading content from the SD card to the Spectrum's memory, providing support for a feature the ZX Spectrum would otherwise not be able to support.

In terms of the possible files that could be loaded into the ZX Spectrum from an SD card, there are a few file formats that are separated into two general types:

- files that simulate an audio tape's structure and data. The most commonly used formats are .tzx and .tap;
- snapshots of the state of the whole ZX Spectrum machine, normally used by software implementations (emulators). The most relevant file formats in this context are .z80 and .sna. .sna files were chosen for being simple in their data organization, whilst .z80 files are more more commonly used.

Since audio tape files hold audio tape data, they require the ZX Spectrum to load them as audio, a process that takes as long as loading an audio tape. This defeats the purpose of faster loading with the SD card. Emulators tend to allow fast loading of these files. One of them details the process, but alerts that audio tapes with a custom loader do not work, as they skip the LOAD routine in the ROM.[30]

Snapshot files contain various data values of the ZX Spectrum's state, saved when the snapshot was created. Its header contains the register values, as well as the interrupt mode, the IFFs' values, and the border color. Specifics on how the data is stored differ between specific file formats. The file formats' structures are further detailed in Subsection 6.9.2 in Chapter 6. The focus of the remainder of this Subsection is discussing the method used to load the contents of these files into memory and the Z80's registers.

Creating new outputs in the T80 implementation is a possible way to load the file's contents to its registers, through parallel load, since FPGA technology allows these changes. However, a mechanism has to be added to a port of the NIOS to multiplex all the registers, selecting which one the data is for. This adds unnecessary complexity to the ULA.

Loading the Z80's registers with assembly code is another way of loading register values, and it uses mechanisms that are already in place. It is done by giving the Z80 the instructions inside an interrupt service routine. As explained in Subsection 2.1, the Z80 has 3 different maskable interrupt modes, with mode 1 being used for the keyboard scanning routine. Mode 0 can be used to send the instructions one by one, but it can prove unpredictable, since after each interrupt, the CPU is likely to execute other instructions if the following interrupt instruction does not start on time. Mode 2 can be used to point to a full routine, however, the ULA's interrupt signal for keyboard scanning would keep constantly triggering it, unless it was blocked.

Loading the memory contents directly implies the use of DMA using the Z80's bus request input. Beforehand, the Z80 needs to "request" that NIOS II start the process since the user selects the file using a menu running on the Z80. NIOS II acts as a peripheral receiving commands from the Z80 to start operations such as "list files" or "load this file". The HALT instruction is used after every command the Z80 sends to guarantee that no other instruction is executed until NIOS II finishes processing the command. Interrupts are disabled beforehand, so the ULA does not release the HALT state. After a DMA ends, the Z80 checks for interrupt requests, guaranteeing that no other instructions are executed before the memory write process is finished. Afterwards, a non-maskable interrupt is triggered by NIOS to resume the processor. This type of interrupt is not disabled with the "disable interrupts" instruction.

Non-maskable interrupts execute instructions starting at 0x0066. By altering the ROM's contents in that address, NMIs make the Z80 perform a jump to an established memory location where the register loading routine is. The routine at address 0x0066 in the original ROM allows for a system reset but it is never used in standard Spectrums, so replacing it does not cause issues. The established memory location for writing the register-filling routine is the screen memory region. The routine is written via NIOS after the file's data contents are written to RAM because an interrupt is required to resume the processor regardless, so using it to load the register values is more efficient. The location of the routine guarantees that none of the file data in RAM, which could be important code for the software, is replaced. The NIOS reads the file and extracts the register values required to generate the routine and load it into screen memory. This causes extremely brief visual artifacts, but NIOS detects when the routine has finished execution and overwrites it with the original video data with a DMA, promptly restoring it.

5.6.1 Hardware

The NIOS II processor performs the low-level operations of the SD card, such as the initialization procedure and the commands to read data blocks from the card. The software is written in C. The target platform's System CD has an SD card demonstration that uses its own library to read or list files. However, it lacks write functionality, so save state functionality would require a different SD/FAT library. The NIOS II processor is also configured with the appropriate I/O to connect to the SD card as well as the ZX Spectrum's control busses.

5.6.2 Software

The selection of files present in the SD card on the ZX Spectrum requires a menu to list those files, with the possibility of selecting one of them to load its contents onto RAM. A routine was developed containing the menu and the instructions to communicate with the NIOS II as a "peripheral" to read the SD card.

Similar to the ZX Spectrum 128k models described in Section 2.7, an initial menu appears on boot that has the option to load from the SD card directly or to open the normal 48k's BASIC editor. The SD card loading option makes the file list menu appear and allows the selection of a file from the list. Both menus are made with the same routines and mechanisms used for the ZX Spectrum 128k's menu. The contents of the file menu are written to memory by NIOS II after the "get page" command is received. The state of the page and the highlighted file is saved in memory. The command to load the selected file sends this information to NIOS II, which it uses to identify and load the file.

All of this code could fit into the Spectrum 48k's ROM, seeing as it possesses 1170 bytes of unused space at address 0x386E. If space ends up being an issue, NIOS II could write extra code in a defined memory address and the code in ROM could point to it. The menu only needs to exist when the computer turns on for the first time, so subsequent programs overwriting the menu is not an issue, as long as the NIOS II writes the routine in the same memory region with every reset. The filenames however need to be updated in memory via NIOS for each new page request, making the menu's formation dependent on NIOS.

In the original 128k ZX Spectrums, the 48k basic menu option switches the ROM page to the 48k's ROM and executes that code. In this case, the custom menu is placed in a 48k ROM, so initializing the normal BASIC editor is done by resetting the machine and

preventing the menu from loading by verifying a flag in memory. This is done with the knowledge that a region of RAM is not reset if the NEW command routine is used. This routine only resets RAM contents from the variable RAMTOP [0xFF57] down to 0x4000, as opposed to the START routine, located in address 0x0000. 0xFF58 is the start address of the reserved memory region in RAM, as illustrated in Figure 2.4. [35]

All registers have load instructions and register A is no exception, but the other register in its pair, the F register (flags register) cannot. This poses a problem since the flags are an important part of the machine's state. An idea of how alter the flags register was to generate code that would purposefully manipulate it, but this could prove to be a lot of work just for setting flag values, and its size could be unpredictable. As seen in the Z80 CPU's documented opcodes[21], PUSH and POP instructions exist for the AF register pair. Therefore, the flag values could be loaded simultaneously with the A register if these values were written into the stack and POP'd directly into their registers. After this solution was thought up, loading the rest of the registers by using only the stack was considered. However, that would add a lot of values to stack and, since the stack could be located in different addresses depending on the software, it could theoretically overwrite important code. AF and AF' only decrease the stack pointer by 4, which is safer.

5.7 Audio

In the original ZX Spectrum, there was a beeper that worked with 1 bit of audio input. In the target platform, the analog audio is output by a 24-bit DAC (serial).

5.7.1 Output

To pass the audio output by the ULA through the DAC, the one bit output had to be extended. By sending the value directly to the DAC's data input, at one of the Coder/Decoder (CODEC)'s configurable speeds, 24-bit values are formed by the one bit of audio. Only two sounds were possible in the original ZX Spectrum, so the only relevant bit to differentiate them in the CODEC is the Most-Significant Bit (MSB). Sending the 1-bit value constantly can lead to values such as 0xFFFFFF or 0x000000 ideally. For value changes in the middle of a transmission, the CODEC receives, for example, 0xF80000 or 0x01FFFF, which is interpreted as a 1 or a 0 respectively. Seeing as the chosen frequency for the CODEC is 18 MHz, this different value is only output for an instant and the sound it produces is not humanly audible.

5.7.2 Input

The input is treated differently since the samples are 24 bits long coming from the CODEC and must be transformed to 0s or 1s. This is done by using the MSB, splitting the full 24-bit value into 2 possibilities:

- values higher or equal to 0x800000 could be interpreted as 1s;
- values lower or equal to 0x7FFFFF could be interpreted as 0s;

The idea is similar to the output, but it requires a more complex mechanism since this data is sent serialized. The MSB is guaranteed by a counter running at the same frequency as the data being received. This bit is then sent to the EAR port in the ULA ports IP block.

5.8 Optional Features

The optional features are internet access and save states.

Internet access refers to the possibility of the implementation using a plugged Ethernet cable to fetch software from a specific URL. The files downloaded would go through the same process as files in the SD card to load their contents.

Save states refer to the creation of new snapshot files, saving the state of the machine the moment it is triggered. These would create a new file for each save state. For this to work, on the SD card side, functionality to create files in the determined filesystem would be required. There would have to be a command done to NIOS to trigger the save state operation, followed by the extraction of memory and the register values to form the file format of the snapshot. It would require two phases:

- memory is extracted, a routine is written to memory for NMI to trigger its execution, and then the interrupt is triggered as the DMA is stopped;
- the NMI routine is executed, writes the values of the registers somewhere in memory, and HALTs. NIOS obtains them and puts back the overwritten memory.

However, when considering the AF and AF' registers that lack load instructions, as stated in Subsection 5.6.2, this order would not work. If the memory is extracted before the routine pushes the values for AF and AF' to stack, then these values are not going to exist in the generated file. Similarly, the SP would not be pointing to the PC, as

required for .SNA files, since an NMI would not have been triggered yet. Therefore, the correct order would be to save the registers first, followed by saving the whole memory and writing it to the resulting file. The memory overwritten by the "save registers" routine would need to be saved before to avoid incorrect visuals.

5.9 Summary

Multiple ideas were considered for the implementation in this chapter. The "T80" VHDL implementation was chosen for the CPU. A resolution of 1024x768 was defined for the VGA output, along with the required clock and timings. A video component separate from the ULA was conceived, responsible for the video data reading and the output directly to the VGA DAC. The main idea behind implementing PS/2 keyboard support and audio was determined. NIOS II was chosen to handle the SD card functionality, reading files from it and writing their contents to memory and the T80's registers. The NIOS also writes menus to memory for user interaction through the ZX Spectrum's T80, adding functionality that would not be possible in an original ZX Spectrum. The details of all these components' implementations appear in the next chapter.

6

Implementation of the ZX Spectrum Fusion

This chapter explains the implementation of the modernized ZX Spectrum on the DE2-115 FPGA board in detail. The components were organized as IP cores so that they could be instantiated in the final system. The main components were implemented separately to ensure they worked properly standalone. A block diagram of the implementation can be seen in Figure 6.1. Each component is described in the different sections below, in generally the same order as they were in the previous Chapter. The internet access optional feature is omitted for not having been completed.

6.1 T80 and Memory Setup

As mentioned in section 5.2, the CPU that is used is the T80, a VHDL implementation of the Z80. This Section details how the T80 was connected to other modules.

6.1.1 System Connections

The T80 was connected to various memory models and the global system's reset counter. This counter was initially made to satisfy the Z80's synchronous reset requirements, which is stated to require up to 3 clocks.[21] This counter begins when the system

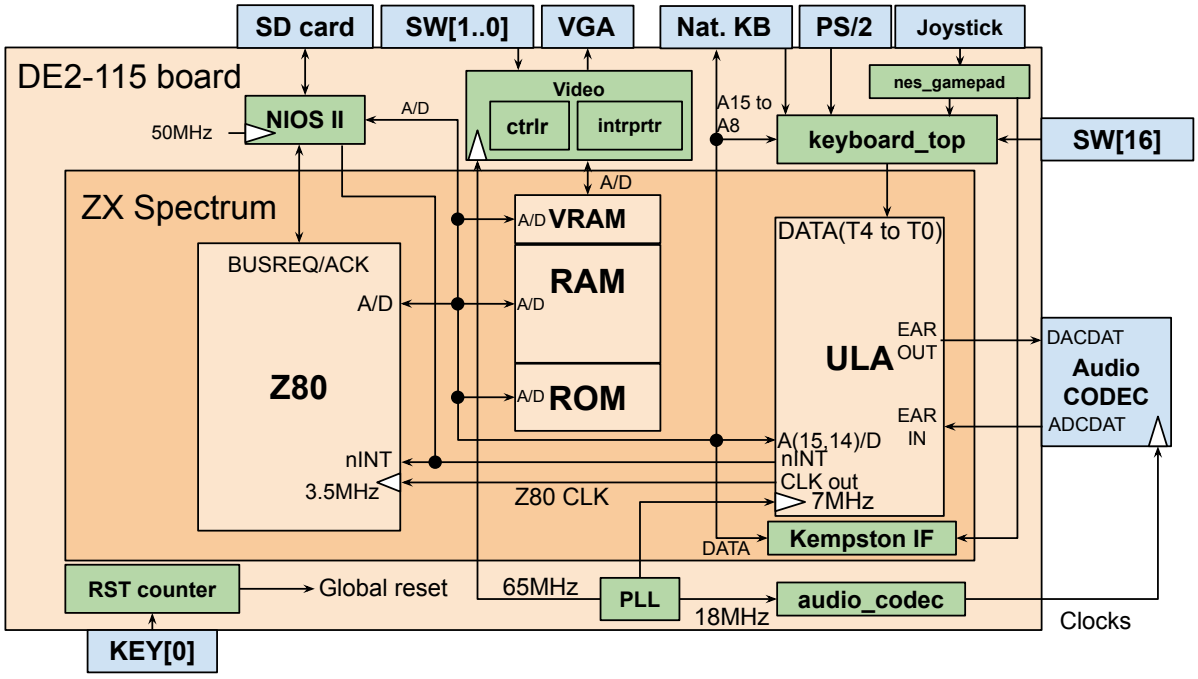

Figure 6.1: A block diagram of the complete system as it was implemented

turns on to guarantee that the T80 is in an initial state. The counter resets by using a button on the target platform, effectively resetting the system.

Before the inclusion of the ULA, a 3.5MHz clock was fed to the T80 by using the PLL. Memory sets were instantiated in the project, including a ROM, video RAM, color RAM, and remaining RAM. Signals were created based on the states of MEMREQ, RD, and WR, as well as the address bus, to detect when each component was being accessed:

- **ROM**: Accessed when MEMREQ and RD are on and address bits 15 and 14 are at 0 (>0x4000);
- **Screen data RAM**: Accessed when MEMREQ is on and the high bits of the address are either "0100" (0x4000 to 0x4FFF) or "01010" (0x5000 to 0x57FF);
- **Color data RAM**: Accessed when MEMREQ is on and the high bits of the address are "010110" (0x5800 to 0x5BFF) and the bits 9 and 8 are not both at 1 (limiting the maximum number to 0x5AFF);

The RAM enable, being more complex, is not listed with the other memories. Its logic function is:

$$!MREQ_n.(A15 + (A14.(A13 + (A12.A11).(A10 + (A9.A8))))) \tag{6.1}$$

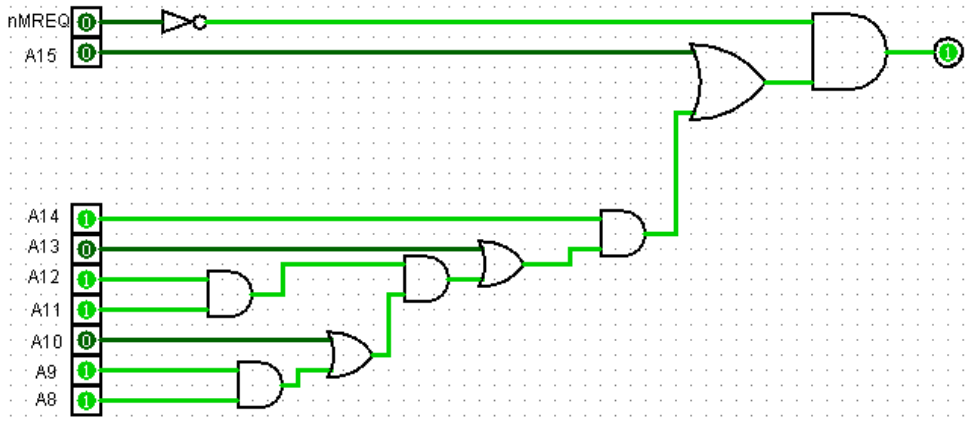

Figure 6.2: Logisim design of the RAM enable expression/circuit, with 0x5B set on the address bits

This expression was based on the range of addresses the remaining RAM occupied, which is 0x5B00 to 0xFFFF. Due to the complexity of the expression, Logisim (a simple digital circuit design and simulation tool) was used to more easily visualize and conceive of the expression that would result in this range. The design that corresponds to the expression can be seen in Figure 6.2.

A data bus input switch for the CPU was created that, based on the access being made, would connect the selected component's output to the data bus for the CPU to read. The T80's interrupt signal is directly connected to the ULA (rather, the stand-in for the ULA, discussed in Subsection 6.2.2).

6.1.2 I/O Interfacing

Connections were created in the top.vhd file as the peripherals were added. They allow the Z80 to interface with peripherals such as the ULA, NIOS, and the Kempston interface. The I/O access enable signals for these peripherals consist of:

- **ula_en**, the enable signal for the ULA, which is on when IORQ is on and the address being accessed is 0xFE;
- **nios_en**, for the NIOS' peripheral communication as discussed in Section 5.6. It is address decoded to values that have been confirmed to not be in use:
  - 0x17: Initialization (detailed in Subsection 6.10);
  - 0x19: Save State functionality;
  - 0x1B: SD card file access (get a file list or write file contents to memory);

  - 0x1D: Online Access functionality.

- **kemp_en**, the enable signal for the Kempston interface, which is on when IORQ is on and the address being accessed is 0x1F.

Multiplexers were added to the top.vhd that initially connect the global address, data, and control busses to the Z80, but when BUSACK is enabled, connect the NIOS' busses instead (nios_ctrl_bus, nios_address, nios_data). This was done to support DMA with NIOS when it was implemented, discussed later in Section 5.6. The control bus output by the NIOS instance has 4 bits that consist of the active-low signals for read, write, memory request, and I/O request.

6.2 ULA

In the previous chapter, in Subsection 5.1, a replica of the original ULA was deemed unnecessary because it included logic for obsolete video output. A proper video component now processes the video, reading the screen and color data at a higher clock frequency than the ULA.

Besides video output, the functions the ULA has are:

- timing the keyboard scanning routine;
- timing flash frequency for attribute blocks with the "flash" property;
- preventing simultaneous accesses to lower addresses of RAM between the Z80 and itself (no longer technically required, but should be simulated for accuracy);
- interacting with the Z80 as a peripheral when it is required to set the border color, output sound, receive keyboard data, or tape data.

These functions are carried out by two IP blocks inside the ULA component: "ula_port" and "ula_count". They are described in these next subsections.

6.2.1 The "ula_port" IP Core

This component, created by Mike Stirling for his ZX Spectrum on a DE1 board project, serves as an interface between the Z80 and the other components.

It works as the peripheral interface of the ULA, used by the Z80. It is enabled when an I/O access is made to the ULA's port (any even address, but normally only 0xFE

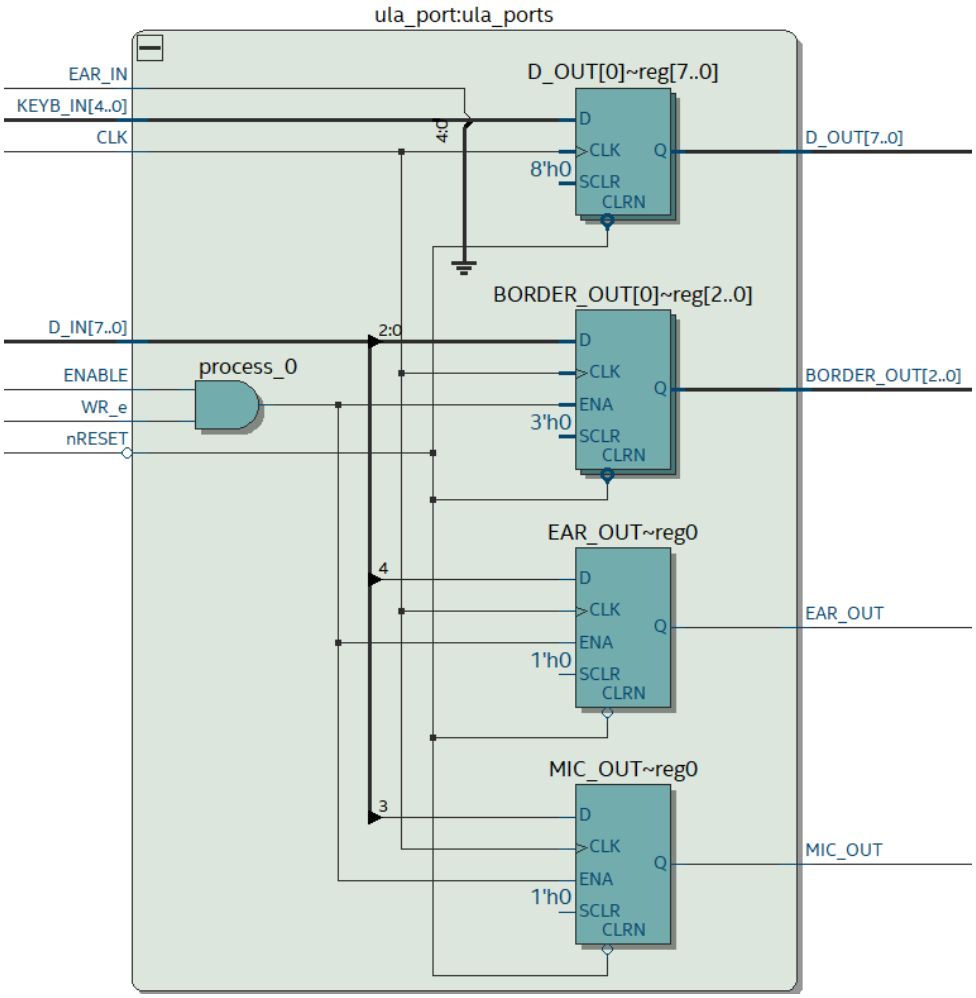

Figure 6.3: The "ula_port" component, in block diagram

is used). By default, its data holds the keyboard and EAR audio tape input signal, obtainable with a read of the "ula_data_out" signal. In write operations, the Z80 can write to the MIC audio tape output, the speaker, and the border color register, in the format described in Table 2.1, in subsection 2.2. A block diagram of this component can be seen in Figure 6.3. It shows the registers the Z80 can write to when accessing the ULA. This component can receive the Z80's data through D_IN, the audio input data through EAR_IN, and the keyboard data through KEYB_IN. The diagram also shows the following outputs: D_OUT for the Z80's read, BORDER_OUT for the video component, and EAR_OUT and MIC_OUT for the audio output.

6.2.2 The "ula_counters" IP Core

This component was made to recreate the correct timings for interrupts and CPU clock stops, i.e. contention control. It holds the "main" horizontal and vertical counters of

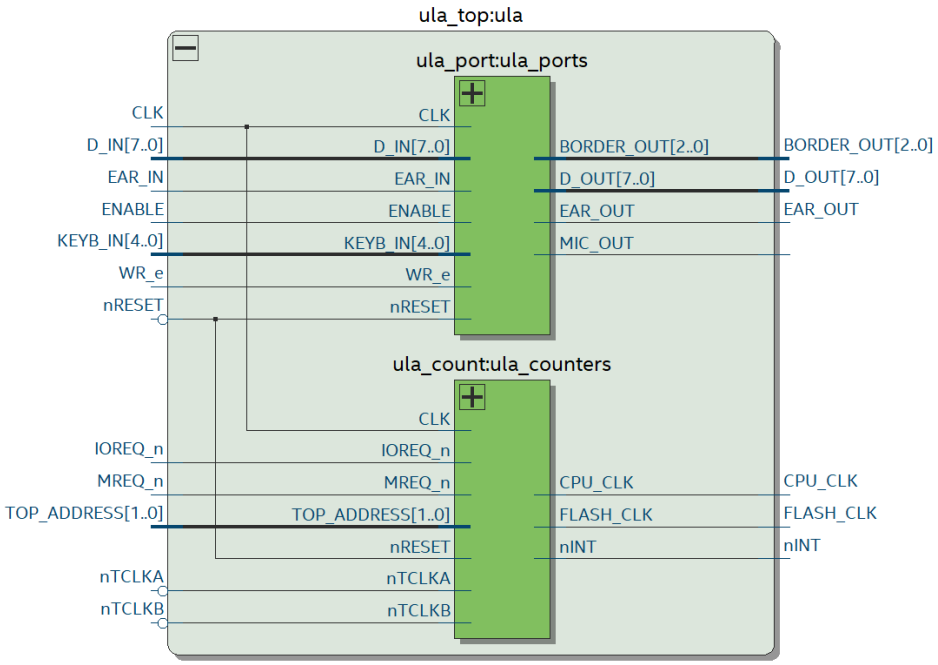

Figure 6.4: The components inside the ULA component

the original ULA, as well as the contention circuit present in Issue 3 ULAs.

A ripple counter is implemented to recreate these counters appropriately, as was mentioned in Subsection 2.3.4. It has a delay between the source clock and each flip-flop switching, due to the output of the previous flip-flop feeding the next one's clock. The flip-flops were created separately, instantiated, and connected in this component to create the counters based on diagrams in Chris Smith's book.[27] A diagram of the component can be seen in Figure 6.4, together with the "ula_port" component inside the ULA.

6.3 Video

This section describes the implementation of a module that can read video data and render it in a VGA monitor. It starts with the controller for outputting in VGA, followed by the module that processes the video memory, and finally, how the borders are drawn.

6.3.1 VGA Controller

As introduced in Section 4.3, VGA signals require specific timings for proper interfacing with the monitor. These timings all have to be configured with the resolution

Horizontal timing (line)

Polarity of horizontal sync pulse is negative.

Scanline part	Pixels	Time [µs]
Visible area	1024	15.753846153846
Front porch	24	0.36923076923077
Sync pulse	136	2.0923076923077
Back porch	160	2.4615384615385
Whole line	1344	20.676923076923

Vertical timing (frame)

Polarity of vertical sync pulse is negative.

Frame part	Lines	Time [ms]
Visible area	768	15.879876923077
Front porch	3	0.062030769230769
Sync pulse	6	0.12406153846154
Back porch	29	0.59963076923077
Whole frame	806	16.6656

Figure 6.5: The timings for VGA 1024 x 768 @60 Hz

of 1024x768 in mind and require a controller to manage the pulses. It is necessary to configure the timings based on VGA standard. The specific timings can be seen in Figure 6.5[40]. The VGA controller that was used was written by Scott Larson [39], and it is written in VHDL.

6.3.2 ZX Spectrum Screen Data

The "data_interpreter" module draws the image generated by the ZX Spectrum on a screen.

Two different BRAM sets store the video memory, separated into screen/pixel memory and color memory. They are separated to have both types of data fetched simultaneously by the video component on time for the VGA. These modules are mapped to the addresses of the corresponding regions in the original memory map of the ZX Spectrum, as seen in Chapter 2, in Figure 2.4.

This module computes the addresses of the color needed for the current pixels being rendered and subsequently sets them. In the original ZX Spectrum, the ULA contained a vertical counter and a horizontal counter. They were used to determine the display and attribute addresses as the screen was being rendered [27] (chapter 15). Individual counters were combined in this implementation to calculate the addresses of the current pixel to render:

- current **x coordinate of real pixels** in a ZX Spectrum pixel, ranging from 0 to 3;
- current **ZX Spectrum pixel** in an attribute block's row of pixels, ranging from 0 to 7;
- current **attribute block (column)** in a row of attribute blocks, ranging from 0 to 31 (counts for each row of real pixels drawn);
- current **y coordinate of real pixels** in a ZX Spectrum pixel, ranging from 0 to 3 (increments after a full row of real pixels has been drawn);
- current **ZX Spectrum pixel** in an attribute block's column of pixels, ranging from 0 to 7;
- current **attribute block row** in a group of attribute block rows, ranging from 0 to 7;
- current **attribute block row group** in screen, ranging from 0 to 2.

These counters were created separately due to the screen's organization. Each counter has its own "weight" due to the organization of the data in memory. For example, each subsequent pixel row in the screen is offset by 0x100 bytes in the ZX Spectrum's screen memory. To match that "weight", the pixel row counter's current value is multiplied by 0x100 (shifted left by 8) and added to form part of the address calculation. The calculation used for the display address is:

$$DISPLAY\_ADDR = curr\_attribute\_column + (curr\_pixel\_row \times 0x0100) +$$

$$(curr\_att\_row \times 0x20) + (curr\_att\_row\_group \times 0x800)$$

The color data is organized linearly in the color memory area. This means that each subsequent attribute block's color is located in the subsequent byte. The address of the current attribute data is obtained with this calculation:

$$ATTRIBUTE\_ADDR = curr\_attribute\_column + (curr\_att\_row \times 0x20) +$$

$$(curr\_att\_row\_group \times 0x100)$$

The relevant data for the primary colors (RGB), in the color data for an attribute block, is organized as 1 bit per primary color, indicating if it is on or off. The brightness of the attribute block is also relevant and is output as a single bit for all three colors. The

"video" module, the video component's top entity, generates these signals. This was done to simplify the "data_interpreter", abstracting the detail on what specific 8-bit color value is used from the component that interprets the screen data. For each of the primary colors, if bit 0 is low then that color is not active, but if it is high, the brightness bit is checked to decide if that color will be of the bright variety (chosen as the value 0xB2) or not (chosen as the value 0xE6). These values were picked for being easily distinguishable while closely matching the original ZX Spectrum's look.

The flash clock comes from the module "ula_count" and is fed into the "data_interpreter". When deciding the color output, depending on the flash clock's bit, the colors change if the attribute data says it is a flashing type. For instance, if the clock is at 1, the opposite color should be flashing, so the ink color will be the paper color and vice versa.

The color data must be ready in time to draw the pixels. In the original ZX Spectrum, when calculating the video addresses, the first 3 bits of the horizontal counter were not used. Instead, these were used to determine the timing to fetch new data. In this context, with a faster video output timing, color and display data must be read in one clock cycle. The clock for the video memory blocks has the opposite polarity of the pixel clock, meaning the data could be obtained half a clock before the next pixel is rendered. Whenever the pixel clock is low and nothing is being rendered, the data for the next pixel is fetched and set before it is rendered. However, the use of the aforementioned counters complicated the address calculation for the next pixel's data. At worst, all the attribute blocks would be offset by 1 in the rendered screen. The effect was mitigated and rendered attribute blocks were mostly correct, except for a 1-pixel wide column which would have outdated data. The first pixel column of the current attribute block was being rendered as the first pixel column of the previous block, as pictured in Figure 6.6, generated with the memory dump mentioned at the beginning of this Subsection.

This problem was corrected by using the row and column values output by the VGA controller directly instead of using the aforementioned counters to keep track of the pixels being rendered. These values can be separated into variables similar to the counters. This computation can be seen in Figure 6.7.

The values resulting from the computation are concatenated to directly obtain the addresses, like so:

$$PIXEL\_ADDR <= group\_num \,\&\, pix\_row\_num \,\&\, att\_row\_num \,\&\, att\_col\_num$$

$$COLOR\_ADDR <= group\_num \,\&\, att\_row\_num \,\&\, att\_col\_num$$

Figure 6.6: The glitch caused by the counters in the old data interpreter implementation (most visible in character sprites, since these can exist between attribute blocks)

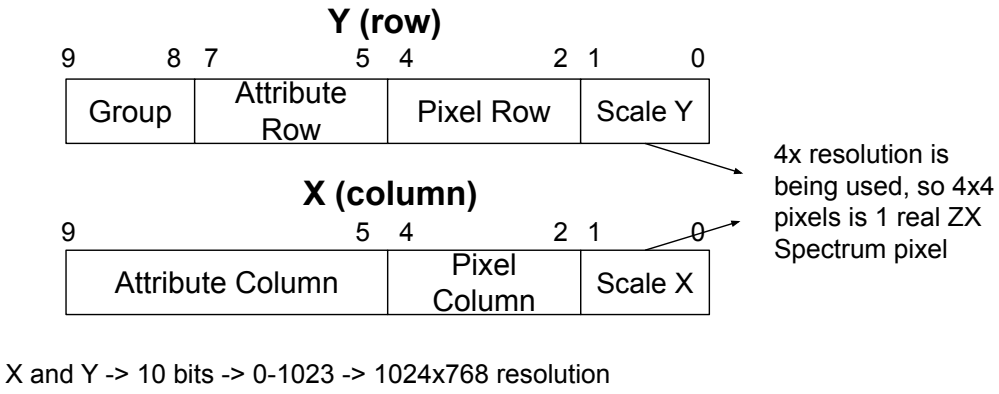

Figure 6.7: The values that can be computed from the row and column numbers

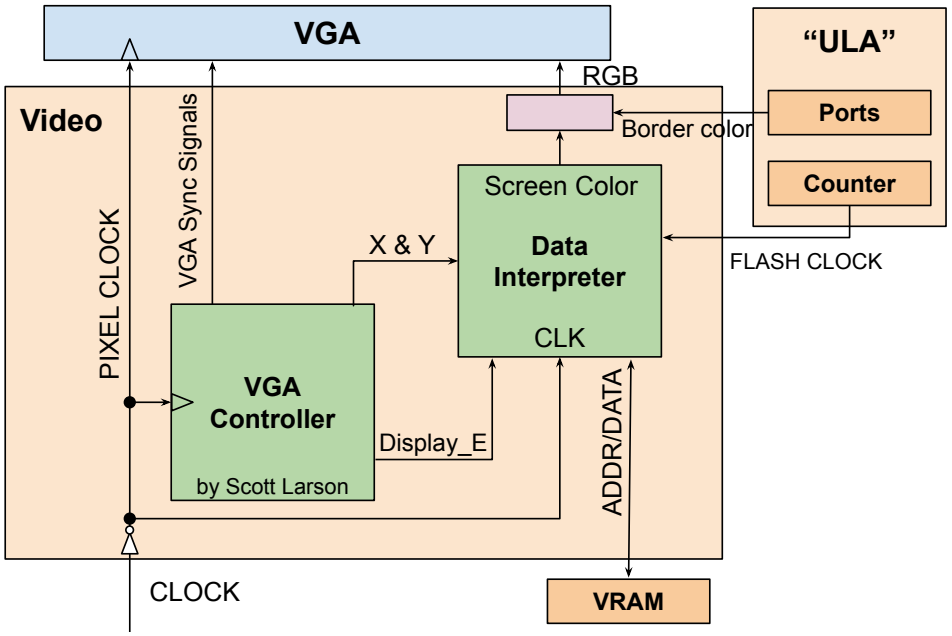

Figure 6.8: A block diagram of the video component

This guaranteed the proper synchronous behavior of the memory address to read in time with the VGA drawing. The full video module is displayed in Figure 6.8, reflecting the previously mentioned change.

6.3.3 Screen Border

The border drawing is handled directly by the top entity video component "video". Limits were defined for when the outputted colors originated from the data interpreter component (active ZX Spectrum screen) or if they were the border color received from the ULA.

The original idea was to have a 1-pixel-wide border in the 4x resolution ZX Spectrum screen (as mentioned in Section 5.4) that was now possible with the VGA's resolution. However, having other sizes for the border, and consequently, the main screen, allows the user to choose if they would want a border more similar to the original ZX Spectrum or not. Therefore, a 2x resolution option, where each dimension of the original resolution is multiplied by 2, allows for a thicker border that resembles the original one. A 1x resolution option is also present for a 1 to 1 experience, despite looking very small in a VGA monitor running at 1024x768. The option serves as a visual comparison between the VGA's resolution and the native ZX Spectrum's resolution.

In the system's implementation, constants were created and kept in "constants.vhd". Some of these include the initial X and Y coordinates for the active screen for each

border setting, guaranteeing that it would be centered. For 2x, the initial X and Y values are 256 and 192 respectively. For 1x, these values are 374 and 288. A visual representation of these values and their relation with the resolutions can be seen in Figure 6.9. In the "video" module, the constants are multiplexed based on the board's switches. The outputs of the multiplexer are compared to the row and column values produced by the VGA controller to decide whether to output the border color or active screen data. "Less or equal" and "More or equal" are used, so that the 4x init values, (0,0), result in a 1-pixel-wide border. The initial values and the current resolution mode are sent to the "data_interpreter", where the difference between the initial values and the current ones (x and y) is calculated to get a proper address value. For example, the top left pixel is no longer (0,0) in 2x resolution, it is (256, 192), so when x and y equal these values, subtracting the "init" values results in (0,0) for the address calculation, thus the fetched data corresponds to the top left corner of the Spectrum's screen.

Then, as described at the end of the previous Subsection (Subsection 6.3.2), values are computed from the row and column numbers. However, the regions where they are taken from differ due to the resolution changes. With 2x resolution, for instance, the address number should only change after 2 pixels instead of 4. This is what the "Scale" values represent in Figure 6.7. In this resolution mode, these scale values are shrunk to 1 bit and the other values are shifted to the right (pixel column becomes the region of bits 3-1). For 1x resolution, the Scale bit does not exist and all values are shifted by 2, effectively making every VGA pixel correspond to a ZX Spectrum one.

On the DE2-115 board, the border modes are set with SW[1] and SW[0], where "00" is 4x, "10" is 2x and "11" is 1x.

6.4 Keyboard

Keyboard implementation can be divided into 2 main parts that are connected in "keyboard_top":

- "PS2_Controller" which is responsible for directly communicating with the PS/2 keyboard in half-duplex mode;
- "input_receiver" which "converts" the PS/2 data received from the controller to Spectrum keyboard data or directly feeds the native keyboard data to the spectrum, if one is connected and the option is enabled.

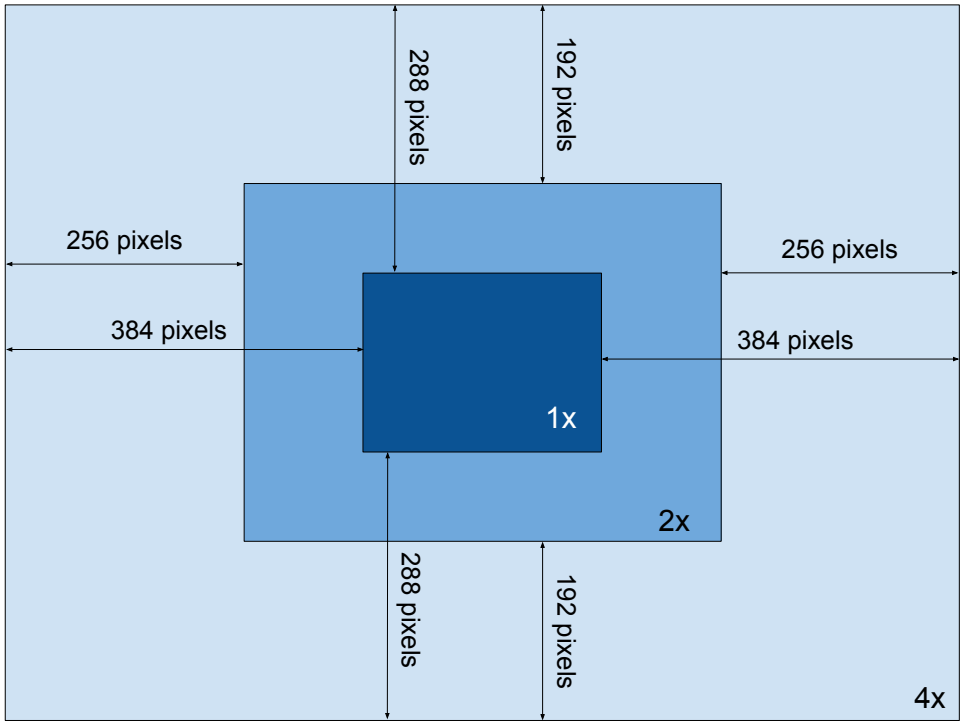

Figure 6.9: The three resolution options and the size of their borders (4x's 1-pixel-wide border is not represented here)

In Section 5.5, the module was denoted by "Keyboard Converter" but in the implementation, "keyboard_top" is the top component that includes the components listed above. It can be seen in Figure 6.10.

6.4.1 PS/2 Controller

The PS/2 Controller by The Computer Engineering Research Group of the University of Toronto[38] was used. This controller was chosen to allow communication with the keyboard in both directions. The controller was written in Verilog. A standalone project was created to test the controller, outputting the key codes that were being

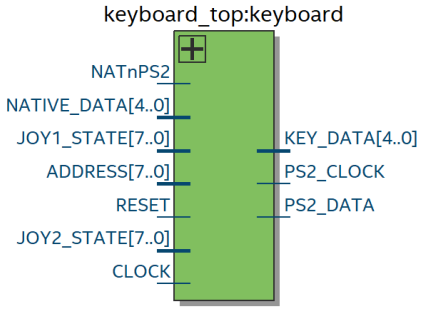

Figure 6.10: The Keyboard Converter component

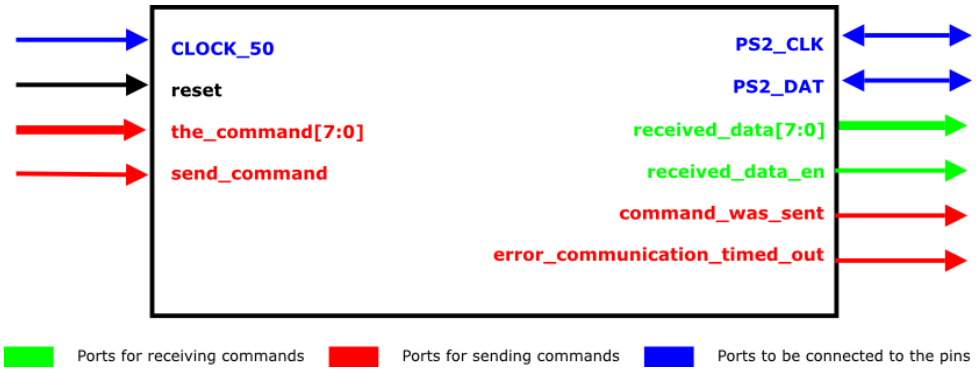

Figure 6.11: EECG's PS/2 Controller[38]

pressed. This standalone project was also used to test commands sent to the keyboard, such as ones that would set the state of the keyboard's LEDs. The controller handles acknowledge responses on its own, sending them as output values. It also serializes host-to-keyboard commands and serializes data received from the keyboard, allowing the Input Receiver to abstract from the interface. Figure 6.11 shows the inputs and outputs of the component. "the_command" receives command data to send to the PS/2 controller and "send_command" is the enable signal for the command. "command_was_sent" and "error_communication_timed_out" output the keyboard's response to the command. "received_data" outputs the data sent by the keyboard and "received_data_en" is the data's enable signal. "PS2_CLK" and "PS2_DAT" are directly connected to the PS/2 port's pins.

6.4.2 Input Receiver

This component is used to receive inputs from the PS/2 Keyboard, joystick, and the native Spectrum keyboard, and output Spectrum formatted keyboard data based on the top address byte it receives. Since the "ula_ports" component is what receives and sends the data to the Z80 when requested, the "input_receiver" component simply outputs the data constantly. This component saves all half-row data in 5-bit signals that represent the real ZX Spectrum keyboard's half-rows based on the keys being pressed and outputs the half-rows depending on the address (KEY_DATA[4..0]). If multiple half-rows are selected, they are combined with an AND, to simulate the original's behavior. An AND gate is used due to pressed keys being active low. The Input Receiver can be seen in Figure 6.12. The "PS2_COMMAND_ACK" input directly connects to the "command_was_sent" output of the controller, notifying that a command was sent successfully. "VALID" is connected to the controller's "received_data_en" output so the input receiver knows when to sample the received data bus (it has "valid" data).

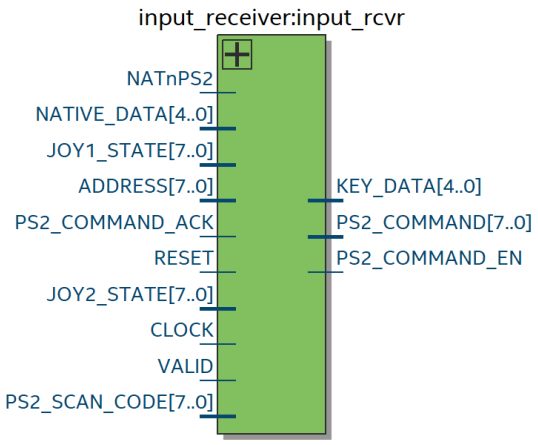

Figure 6.12: The Input Receiver component

The component, through the use of the PS/2 controller, can send commands to the PS/2 Keyboard by using the "PS2_COMMAND" bus and the "PS2_COMMAND_EN" signal. For the PS/2 keyboard's LEDs to be set, dedicated signals are kept to memorize the state of the LEDs themselves, as well as the actual functional state of Capslock and Numlock (numlock, capslock, numlock_led, capslock_led). The LED signals start at 0, meaning the LEDs are initially off, but the numlock signal is on. Whenever the actual Capslock or Numlock keys are pressed, the signals without the "led" suffix update. A state machine was created that initializes any time one of the signals is different from the corresponding LED one, such as the aforementioned numlock signal (starting at 1, but the LED at 0). Its first state sends the command 0xED (Set status LEDs) and changes to state 1. This state waits for an acknowledgment and when it receives it, it sends the argument data for the command, seen in Table 6.1, constructed using the capslock and numlock signals. The next state is triggered and waited on until another acknowledgment is received, finally confirming the change of the LEDs. The LED signals are updated and the state machine's state is set to 0 for the next time a change occurs.

The key press interpretations followed a similar approach as in Mike Stirling's project. This project has a "release" signal to keep track of release codes that were sent by the keyboard and then affect the corresponding half-rows with that signal. The signal is initialized at 0, so any key that is pressed will alter the corresponding half-row data bit to 0, turning it on (active-low). However, when a key is released, the first data byte received is 0xF0 (release suffix, mentioned in Section 4.5), and the next byte identifies that specific key. So 0xF0 causes the "release" signal to change to 1 and the next data received will set the corresponding half-row data bit to 1, effectively releasing the key. "release" is then set back to 0 for the next key that is pressed.

7	6	5	4	3	2	1	0
Always 0	Always 0	Always 0	Always 0	Always 0	Caps Lock	Num Lock	Scroll Lock

Table 6.1: Argument data for setting PS/2 Keyboard's LEDs

This project greatly expands Mike Stirling's keyboard module to include multiple keys and key combinations of the PS/2 Keyboards. Portuguese and US keyboard layouts were created to facilitate the user's interaction and prevent the need to memorize ZX Spectrum key combinations (i.e. Symbol Shift + P = "; US layout: Shift + '; PT layout: Shift + 2). For this, more signals were created to keep track of the states of Shift and Alt. Whenever Shift or Alt is on, a separate switch case in this component is used to interpret the keys differently. For example, the quote key triggers half-row 7, bit 1 for Symbol Shift, and half-row 5, bit 0 for P (Figure 2.9). Key codes are kept in "constants.vhd", excluding Numpad ones which are the same for any layout. Both sets of constants have different values matching the layouts. For example, the double quote key code is equal to the single quote's key code in the US keyboard layout because pressing it with Shift held down results in double quotes, but in the PT keyboard layout it is the key code of "2". The layout can be changed at the design stage by uncommenting the series of lines of the one to use and commenting the other's definitions.

The state of Numlock is saved and the LED indicator is changed so that the number pad acts like a PS/2 keyboard on a computer. Depending on its signal, or shift, the Numpad either acts like arrow keys or numbers. This was done in the same way as the other keys, except filtered by conditions related to the Numlock signal. However, since Numlock and Capslock are toggle keys, they are treated differently. When a release code is received, its state is inverted. The release condition is checked to make sure the states are only set once every press when the user lets go of the key. Otherwise, an unpredictable amount of key presses could be received based on how long a key is being held since PS/2 keyboards send a key's data repeatedly in these conditions.

Sinclair Joysticks translate the activity on their buttons as key presses. Therefore, data from both joysticks is sent to this component. Knowing how the data is organized, it is reconstructed to form new half-rows to correspond to the keys 0-9. The half-rows set by the joysticks are separate signals that are combined with the keyboard ones (through AND gates) to make the final half-rows, guaranteeing simultaneous keyboard and joystick inputs.

The "phantom" key presses were a known bug in the original design and a consequence of the multiplexing used in the keyboard. They were not recreated in this implementation.

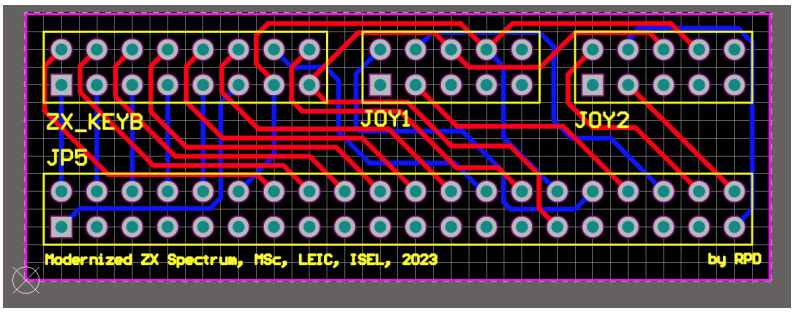

Figure 6.13: Expansion PCB layout

6.4.3 ZX Spectrum Native Keyboard

A native Spectrum keyboard is supported and can be connected through the use of a custom PCB on the expansion header of the DE2-115 FPGA board. The design and schematic of this PCB can be seen in Figure 6.13 and Figure 6.14, respectively.

To process the inputs from a native keyboard, the connections of an original ZX Spectrum were recreated. The top byte of the address bus is output to the native keyboard and its data is read directly. The keyboard component in the FPGA does this if the "native_n_ps2" signal is on, manipulated by switch 16 (SW[16]).

6.5 Audio I/O

The most important aspect to effectively use the audio CODEC of the target platform is to define the signals it needs (described in Section 4.4). In the target platform's system CD lies a Synthesizer example project, written in Verilog, that utilizes a PS/2 keyboard for input and audio input through the use of the audio CODEC. Inside this example, a file named "adio_codec.v" (possibly a typographical error) contains clock generators for the bit clock (BCK) and the stereo clock (LRCK) that receive an 18 MHz clock. These clock signals are sent directly to the audio CODEC. The 18 MHz clock is also sent to the CODEC as its master clock, suggested by its data sheet[41] in the NORMAL MODE SAMPLE RATES subsection of AUDIO DATA SAMPLING RATES. These generators were "ported" to VHDL, to this project, to act as the clock generators.

The bit clock generator in the example uses a couple of constants for its calculation: a reference clock value at 18.432 MHz (equal to the value suggested by the datasheet), a sample rate of 48 KHz (one of the accepted sample rates for the CODEC), a data width of 16 bits and CHANNEL_NUM at 2, for stereo audio. This generator uses these to calculate the number of 18 KHz clock cycles before the BCLK's change. The calculation is as follows:

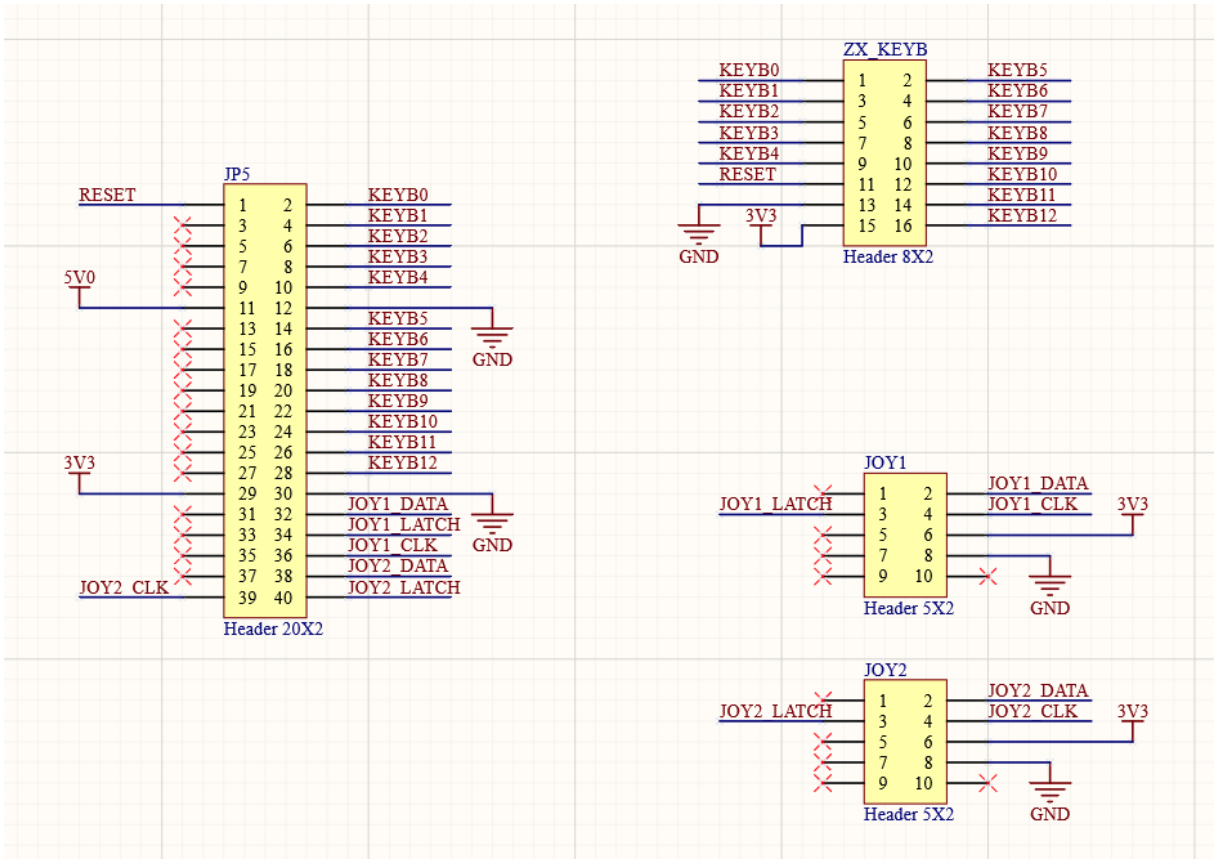

Figure 6.14: Expansion PCB for two joysticks and original keyboard schematic

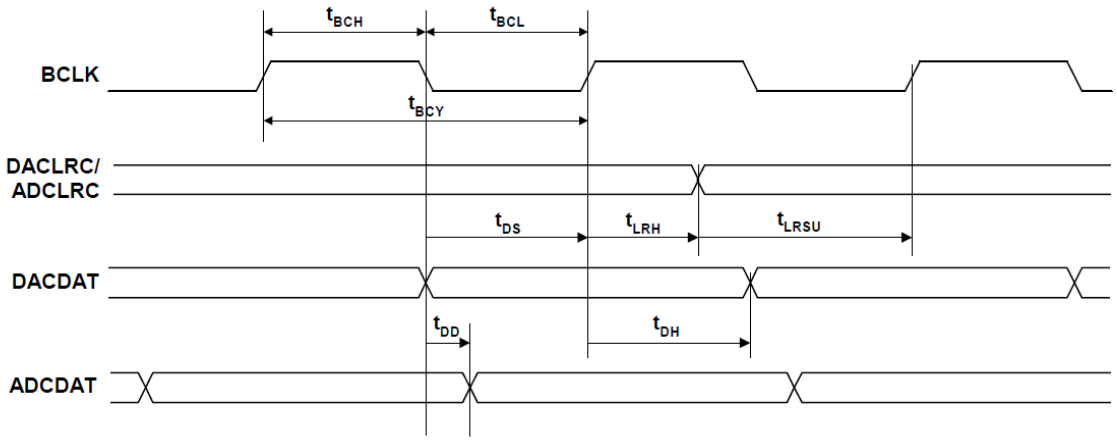

Figure 6.15: Digital Audio Data Timing for WM8731 in Slave Mode

$$\frac{REF\_CLK}{SAMPLE\_RATE \times DATA\_WIDTH \times CHANNEL\_NUM \times 2} - 1$$

In this project, these constants were stored in "constants.vhd", and all values were kept the same except for data width which was changed to 1. This is because the audio output of the ZX Spectrum is done with only 1 bit.

The stereo clock uses only the sample rate in its calculation, in a similar way. The calculation is as follows:

$$\frac{REF\_CLK}{SAMPLE\_RATE \times 2} - 1$$

This results in a lower clock speed for the stereo clock of 96000-1 MHz compared to the bit clock's 192000-1 MHz for a data width of 1. This means that the stereo clock alternates for each bit clock's rising edge. For the stereo clock, alternating between high and low select the left and right audio channels to send data to. An example of the timings can be seen in Figure 6.15.

These calculations, with the constant values inserted, result in 192000-1 and 96000-1 for the BCLK and the LRCK respectively. These are the numbers used for the clock generator counters' limits, to regulate when the polarity is reversed. A diagram of this component can be seen in Figure 6.16. Since these are constants, Quartus synthesizes the verification as simple comparisons with the numbers from these calculations.

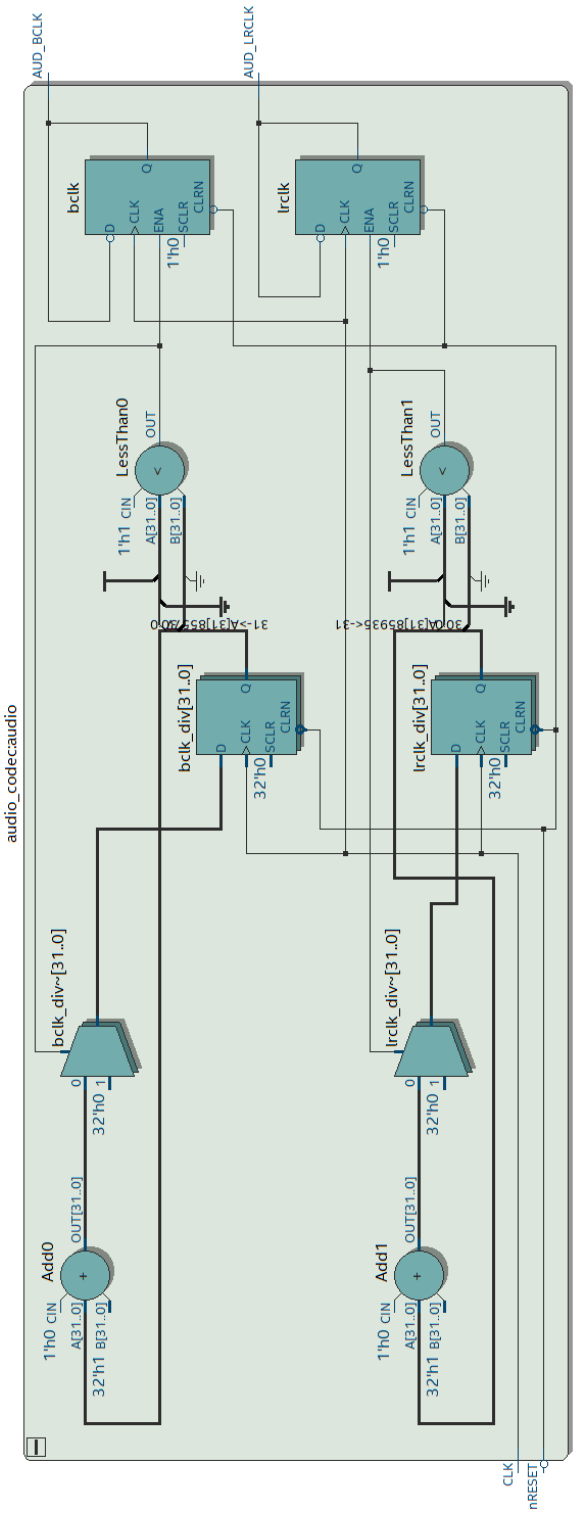

Figure 6.16: Audio CODEC component diagram

6.5.1 Output

For audio output, the SPEAKER OUT bit from the ULA is directly connected to the CODEC's DAC data. The clocks previously described transmit this one bit as 24-bit values that the CODEC can use.

6.5.2 Input

Audio input, as mentioned in Section 5.7, is obtained from the CODEC, as a 24-bit value that needs to be interpreted as a 1-bit value. A component was made called "audio_adc" which contains a 4-bit counter and a 24-bit shift register that receives the data sent by the CODEC. Only the MSB is sent to the EAR input of the ULA, so when the 24th bit is received, the MSB changes and the ULA receives this new value instead of the previous one that was updated 24 bits ago. A diagram of this component can be seen in Figure 6.17.

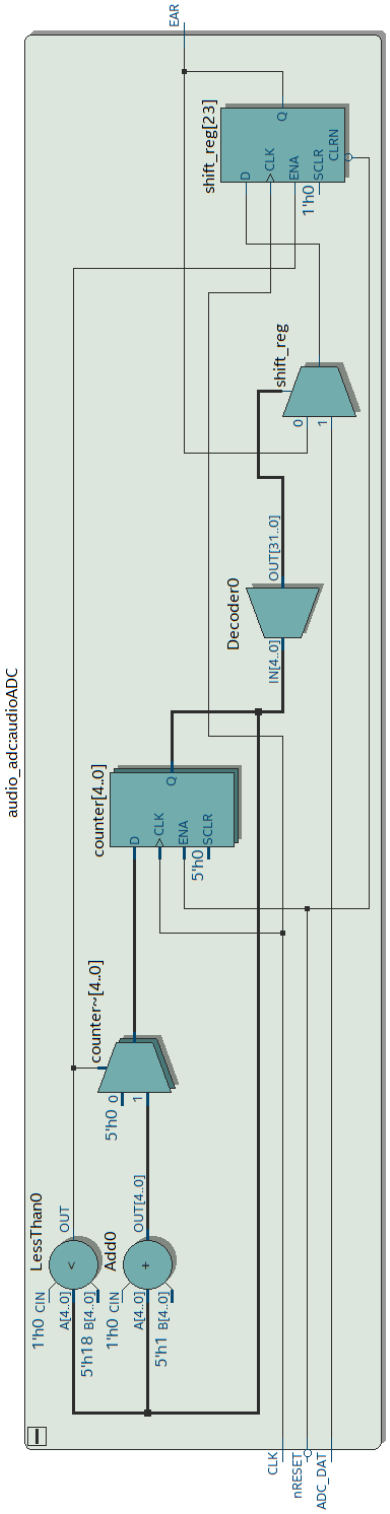

Figure 6.17: Audio ADC component diagram

bit	7	6	5	4	3	2	1	0
button	right	left	down	up	start	select	B	A

Table 6.2: NES gamepad button state order

6.6 Joystick

This implementation supports two joystick interfaces: Kempston and Sinclair. Both of these are handled differently by the ZX Spectrum. Kempston interfaces can only support one joystick, while Sinclair can support up to two. Therefore, two joysticks are connectable to this implementation. Details on the physical controller and how it connects to the board are presented in the following Subsection, followed by the implementation of the Sinclair and Kempston interfaces.

6.6.1 Joystick Controller

Due to being unable to get an original joystick, the ones used in this project are NES gamepad clones with DB-9 connectors. To connect these controllers to the target platform, an expansion board was used. This board was previously introduced in Subsection 6.4.3.

The state of a controller's buttons is sent from an 8-bit static shift register inside it when this data is requested by the target platform by enabling the load/latch signal, followed by 8 clocks for the 8 buttons. The Least-Significant Bit (LSB) is sent first, and the order of each button's state after a full transmission can be seen in Table 6.2.

To capture this, an IP core was used ("nes_gamepad.vhd"), made by Rui Policarpo Duarte that was created for a previous project. This core, based on the clock it receives, keeps a clock counter for the controller, to time its frequency with the capture of each bit of data (button state). It saves this data and sends it as an output, deserializing the state the controller sent. This data is used to implement both the Sinclair and Kempston interfaces.

This IP core was modified to receive and output the data for each controller separately, whilst outputting the latch and clock signals shared by both. This altered block can be seen in Figure 6.18. Since Kempston interfaces only support 1 controller, the option to switch between interfaces only applies to controller 1, with controller 2 being Sinclair only. This was done with a simple multiplexer that is selected using SW[17]: 0 for Sinclair and 1 for Kempston.

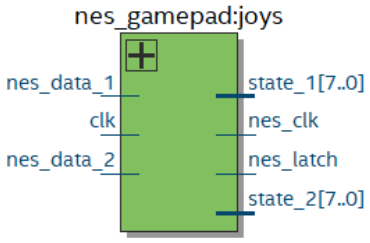

Figure 6.18: The component for deserializing the NES style gamepads' data

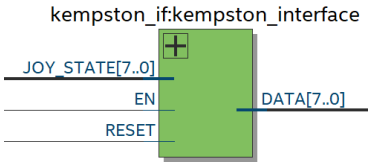

Figure 6.19: The component for the Kempston interface

6.6.2 Sinclair Interface

The controller's states are sent to the keyboard component, to the input receiver, and processed as the keys 0-9. These keys occupy 2 half-rows so an auxiliary 5-bit value is saved for each row and computed from the controller's state. These auxiliary values are then combined (with AND gates) with the keyboard-specific ones to form the final half-rows. Since the NES controller has extra buttons, these were mapped to extra keys of the keyboard:

- Q W E: Controller 2's SELECT START and B
- I O P: Controller 1's SELECT START and B

These were added with the idea that, in games that allow custom controls, the player could map keys to these extra controller buttons, making use of them.

6.6.3 Kempston Interface

A simple component was made consisting of latches that are enabled when an I/O request is made on port 0x1F, the Kempston interface port. These latches are set up to "organize" the controller state bits from their received order (shown in Table 6.2) to the correct data format expected from the ZX Spectrum (shown in Figure 2.17). This component can be seen in Figure 6.19.

6.7 Reset Circuit

To reset the whole system, a reset counter was created. This counter is first mentioned in Section 6.1 in the context of performing the synchronous reset of the T80. Since some components have clocks that derive from others, the order each one is reset had to be considered. For instance, the Z80 receives its clock from the ULA component, so if both are reset at the same time, the Z80 will not be clocked to perform its reset. Therefore, a reset/boot sequence had to be created.

A counter allows the use of the output number as "checkpoints" for when the reset signal should be on. The counter uses the global clock of 50 MHz. It has 10 bits and the MSBs of the sequence were used to control the reset signals.

Since the PLL sends clocks to most components, this should be the first one to start, so that it can supply the other components with clocks while they reset. This reset signal stays on while bits 9 to 3 are at "0000001". This means that it is enabled for 8 clocks and then never again until the counter is reset. The following components to reset are the video, keyboard, NIOS, and ULA components. These are executed simultaneously since they do not depend on each other, but the CPU's reset must happen after the ULAs. The middle components reset while bits 9 to 5 from the counter are at "00001", whilst the CPU resets when bits 9 to 7 are at "001". These numbers mean that the reset signals will be on for 32 clocks and 128 clocks respectively. These numbers were chosen only for the orders, to make sure they were distinct and far apart.

The reset counter is asynchronously reset by KEY[0] in the target platform (the far right button).

6.8 NIOS-Z80 Interface

This subsection describes the implementation of the interfaces set up between the ZX Spectrum hardware and the NIOS II. These interfaces are:

- **DMA** with the Z80: used to write software and data in the ZX Spectrum's memory;
- A **peripheral control** interface for the Z80: used for issuing commands to NIOS regarding the menu and SD card (potentially the optional features too).

6.8.1 DMA Interface

To issue a DMA request to the Z80, NIOS II sets the active-low bus request (BUSREQ) signal low for the duration that it requires the DMA. The Z80 only samples the input after finishing its ongoing instruction, so access is only truly granted when its bus acknowledge signal (BUSACK, active-low) goes low. Hence, the "DMA_request()" method, present in dma_hal.c, has a maximum number of tries it receives as a parameter and keeps the bus request signal active for this duration while it waits for the BUSACK signal from the Z80.

The functions created to manage this interface allow for interactions to be made with memory modules and I/O. These functions set the control, data, and address busses based on the arguments received, perform the memory or IO access, and then reset the control signals. Functions with "_buf_" in their names work with data buffers to execute multiple reads/writes in a row by using a buffer. These were implemented by calling the one-byte versions of the operation a "len" amount of times.

This interface has two functions to disable DMA:

- **DMA_stop()** stops the DMA normally;
- **DMA_stop_w_interrupt()** stops the DMA with the NMI signal on, to trigger it as soon as the Z80 resumes.

The latter function is used to resume the Z80's HALT state, enabled after every command sent to the NIOS II. This guarantees that the Z80 waits until NIOS finishes processing commands. Only NIOS can resume the Z80 in these situations because maskable interrupts are disabled before commands are sent. This function is also used for the Z80 to execute the register-loading routine after a file is loaded. Due to the NMI signal being on when the DMA stops after the file is loaded, the Z80 jumps directly to the register-loading routine.

To amend the visual artifacts caused by routines written in video memory by NIOS (discussed in Section 5.6), an extra function, called "wait_until_routine_ends()" is included in the control of this interface. This function reads the CPU's address bus in parallel for a maximum number of attempts, and checks if it is higher than 0x57FF, the last address of screen memory. This forces the code to wait for the Z80 to finish executing the NMI routine so it can be overwritten.

6.8.2 NIOS II Control Interface

The NIOS control interface is used to receive commands sent by the T80 to NIOS through I/O instructions. By using these instructions, the CPU communicates to NIOS as a peripheral. These commands include:

- Obtaining the list of files in the SD card, per page;
- "Loading" a file from SD card, identified by the current page and its index in the page;
- For save state functionality, creating a snapshot based on the system's state;
- For online access functionality, obtaining the list of files or "Loading" a file, with the same parameters as the SD card commands (future work).

"Loading" files is done by writing its data contents into memory and the register values to the Z80's registers, as described in Subsubsection 6.8.1. The file is identified by the page and its number on this page due to the short space available in the data bus that would not allow filenames. The page number's limit is greatly increased by setting it to the upper address byte (256 possible pages) since peripherals in the ZX Spectrum are mapped to the lower byte of the address. The data bus is used for the game number in write operations to the established port. This bus also has a limit of 256, but only 24 names can fit on a ZX Spectrum screen (32x24 attribute blocks).

Obtaining the list of files is done with the same lower address byte as "loading" files, but with different control signals. This was done to save in I/O addressing space. All commands are mapped to the lower address byte and the read/write signals:

- I/O write on 0x19 trigger NIOS to save the state of the machine into a file;
- I/O read on 0x19 notifies NIOS that the register values have been stored in memory, so it can initiate the second phase of the save state operation, mentioned in Section 5.8;
- I/O read on 0x1B triggers NIOS to write a list of files into memory, where the top address byte corresponds to the page number requested;
- I/O write on 0x1B triggers NIOS to write the contents of a selected file into memory and the registers, where the top address byte corresponds to the page number and the data corresponds to the number of the selected game in that page;

- I/O read and write on 0x1D trigger NIOS to do the same as in the SD card commands, except the source of the files is the internet (future work).

These commands are received by the NIOS through the following signals:

- **cpu_address**: an input connected to the Z80's address bus;
- **nios_en**: the enable signal for the interface, created from the Z80's IORQ signal and the address;
- **cpu_cmd**: an input connected to the Z80's data bus;
- **cpu_rd_n** and **cpu_wr_n**: the Z80's control signals;

The NIOS control interface provides functions to abstract from the command values. These are used in main.c to interpret the command received. The functions are:

- **enum get_if_type()** returns an enumerator value based on the low byte of the address being accessed. This value determines the command type, as introduced in Subsection 6.1.2;
- **int get_page_num()** and **int get_game_num()** return the values in the upper byte of the address and the data bus respectively, contextualizing them to what they mean in this context and turning them into integer values;
- **bool is_read()** and **bool is_write()** returns true or false depending on the control signals.

A register had to be created to store the data of the last command issued by the Z80 to the NIOS II due to their clocks being different from each other, which can be seen in Figure 6.20. The register is enabled when a command is sent to NIOS. That is when the address associated with the NIOS peripheral interfaces is set as well as the I/O request signal and either of the read or write signals is enabled. A function called "void listen_for_en()", in this interface, is used to wait for a command to be available in the register. It has a loop that stops when the enable output of the register is on. NIOS receives this signal through an input port called "CPU_CMD_EN". After NIOS processes a command, it resets the register. The function "void per_cmd_ack()" turns on the output signal "CPU_CMD_ACK" for 1 ms. This signal is fed into the RESET input of the register.

The full I/O of the NIOS instance can be seen in Figure 6.21.

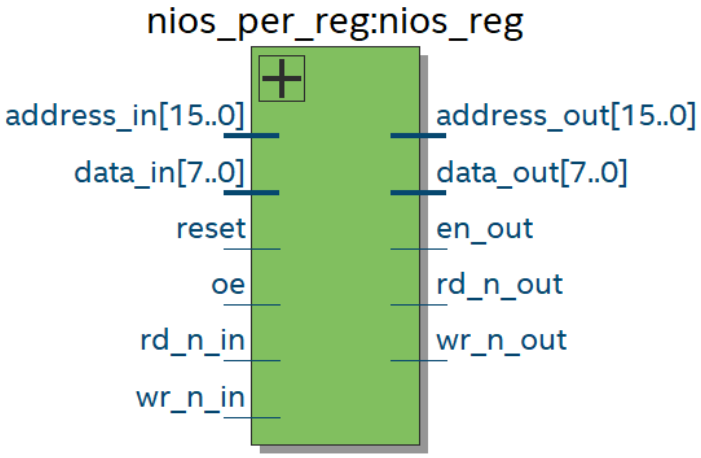

Figure 6.20: The NIOS peripheral interface register

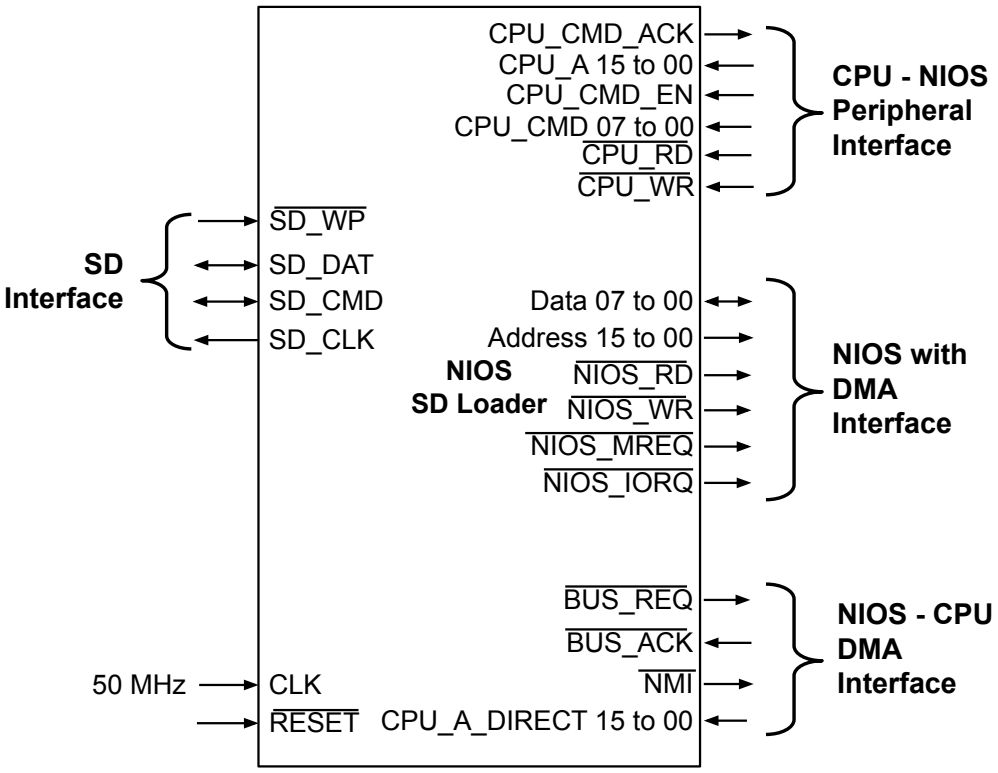

Figure 6.21: The NIOS instance's I/Os

6.9 SD Card

The SD card support implementation was achieved with the help of the NIOS II processor. The details of the implementation are written in the following subsections.

6.9.1 SD Controller

Direct interaction with the SD card was done through the NIOS II, as presented in section 5.6. This Subsection details how the controller was set up.

The System CD for the target platform contained various demonstrative projects that used different peripherals throughout the board. One of them, called "DE2_115_SD_CARD" was specifically for interacting with SD cards. It implements the basic commands SD cards accept and supports reading files in the File Allocation Table (FAT) file system. The code was made to interface the FPGA and the SD card via SD mode, instead of SPI. Inside this software, inside the folder "terasic_fat", the file "FatInternal.h" had an enumerator with the supported partition types, FAT16 and FAT32. However, it only considered FAT32 with Cylinder-Head-Sector (CHS) and Logic Block Address (LBA) addressing (value 11) and not FAT32 only with LBA (value 12), despite the code being made with LBA addressing in mind. The FAT32 value was changed to 12 and a new value was made for FAT32 with CHS addressing, as can be seen in Listing 6.1.

```
1 typedef enum{
2     PARTITION_FAT16=6,       // limited to 2G
3     PARTITION_FAT32_CHS=11, // FAT 32 with CHS addressing
4     PARTITION_FAT32=12       // FAT32 with LBA addressing
5 }PARTITION_TYPE;
```

Listing 6.1: The supported partition types enumerator

In the file FatFileSystem.c, in the function "Fat_Type", a condition was added for this new enumerator value to avoid false errors of the SD card being unsupported in case the user has a FAT32 with LBA addressing. There were other verifications in FatInternal.c that had to be altered to include the other value.

All the files from the DE2_115_SD_CARD folder, inside Software, were copied to the SD folder in the "Software/memory_manipulator" path of this ZX Spectrum implementation's project. "memory_manupulator" is the name given to the software project.

The SD interface was implemented in C to further abstract the main code from the SD card controller. This interface is in the aforementioned SD folder and was made with

the supported file types in mind. Its main function is to read all the files in the inserted SD card to save them in a struct called FILENAMES, consisting of a string array of all the filenames and this array's size. Its other functions use functions from the SD card controller in the System CD example to initialize/close the SD card, as well as access files. Its header file also includes definitions for constants related to established parameters, such as the filename lengths and the files per page on the ZX Spectrum file menu, all based on the size of the screen. Each letter occupies one attribute block, and the ZX Spectrum's screen consists of 32x24 attribute blocks, so filenames can have up to 32 chars. It was defined that a page should hold up to 16 files to allow for some space on the top and bottom edges of the screen for the menu's look. The header file of this interface can be seen in Listing 6.2. FAT_HANDLE and FAT_FILE_HANDLE are part of the SD card controller in the System CD and are void* for structs that are used to keep track of information such as volume info.

```
#define FILENAME_LEN_SD 32
#define FILENAME_LEN 32
#define FILES_PER_PAGE 16

typedef struct {
    size_t size;
    char ** filenames;
} FILENAMES;

FAT_HANDLE init_SD();
bool is_supported_file(char* filename, size_t len);
FILENAMES list_files(FAT_HANDLE hFat);
void close_SD(FAT_HANDLE hFat);

void print_filenames(FILENAMES files, bool free_en);

FAT_FILE_HANDLE init_file(FAT_HANDLE hFat, const char *pFilename);
void close_file(FAT_FILE_HANDLE hFile);
```

Listing 6.2: Header file for the SD card API

6.9.2 File Reader

Loading a snapshot requires reading the file, decrypting it if necessary, extracting its data, and writing the data to the appropriate locations (memory, registers, and ULA). The file_reader folder contains functions written in C to load .z80 and .sna files.

.sna files only support 48k spectrums by default. Some emulators extended the format

to support 128k snapshots, but only 48k snapshots were considered for this implementation. Their headers contain the register values, the interrupt mode (explained in Section 2.1) of the CPU, the interrupt state held by IFF2, and the border color.[28] The raw contents of memory are in the remaining data of the file and always have the same size (48 KB). This format saves the program counter in the stack before the snapshot is generated. Similarly, when an NMI is triggered, the Z80 pushes its current pointer to stack before jumping to the NMI routine. NMI routines always end in a RETN instruction, used to return from NMIs. This instruction pops the address in the stack and jumps to it. Therefore, a return instruction is enough to continue the program in the snapshot.

Despite being the simplest to read (no compression), they are not popular. So, few games are available in this format. This led to the implementation of support for the popular **.z80** format. There are 3 versions of this file format. It supports compression and different ZX Spectrum models.

Version 1 only has one header, which includes the contents of the registers, the border color, the interrupt mode, and hardware details such as if joysticks are plugged in, the ULA issue in use, and the video synchronization's state. It also includes a flag that indicates if the data in the file is compressed. The compression method consists of replacing repetitions of more than 4 bytes with 4-byte codes, using 0xEDED as a tag, followed by two bytes that determine how many times the last byte of the tag is repeated (ED ED xx yy, byte yy repeated xx times). It considers "dangerous" combinations, like two EDs in a row being changed into ED ED 02 ED, or bytes directly following a single ED not being included in any of those tags (ex: ED 00 00 00 00 00 00 is compressed into ED 00 ED ED 05 00) to conform with the established compression method. The end of file for version 1 compressed files is identified through an end marker, 00 ED ED 00. [43]

Version 2 and 3 have an additional header with varying length depending on the specific version. They are identified by the contents of the PC present in the first header, which should be 0 if the file is not version 1. The additional header contains the actual PC value, as well as more specific details about the ZX Spectrum hardware that was being emulated at the time of the snapshot's creation, such as its model (48k, 128k, ...), its plugged in interfaces, its version of the ULA, the contents of the sound chip registers and so on. This last detail is only present in version 3 files. Since this implementation is of a 48k ZX Spectrum, most, of these details are unnecessary. The file version and the ZX Spectrum model used are checked to determine if the snapshot is compatible with the implementation, and where the data starts (since the headers' lengths vary). Version 2 and 3 files are always compressed, and, due to the inclusion of other ZX

Spectrum models, this compressed data is separated into different data blocks, one for each page of memory (16KB each in size, after decompression). Each data block has a small header containing the length of the block, followed by the page number. The compressed data of the block starts after this header. The end marker no longer exists in these versions, since the length of the blocks is enough to discern the end of the file. 48K ZX Spectrum memory is saved as pages 4, 5, and 8, each of them corresponding to the address spaces 0x8000-0xBFFF, 0xC000-0xFFFF, and 0x4000-0x7FFF respectively.[43] These values were determined through the creation of .z80 files on Windows ZX Spectrum emulators and analyzing the contents of RAM concerning the created file.

6.9.2.1 Register Loading

To load any snapshot, it is necessary to load the full state of the machine, and that includes register values. These register values are obtained from the snapshot files, a process detailed next. A struct was created called REGS, defined in file_format_aux.h. This struct was defined to hold all register values, as well as the other details present in the files such as the border color and interrupt state. Despite those not being Z80 register values, they are important for the state of the machine. This struct is used as a parameter for the function "generate_routine". Based on the values of this structure, and the assembly opcodes defined in asm_opcodes.h, the function can put together an assembly routine, byte by byte, and return it as an array of bytes.

The created routine uses the A (accumulator) register as an intermediary to load the values to each register since not all the registers have instructions to load immediate values to them. It starts by setting the border color by writing it into the address of the BORDCR system variable (LD (addr), A). Next, it loads the values of the R and I registers (in the case of the I register: LD A, i_value; LD I, A). After this, the auxiliary register values are loaded onto the normal registers (HL' => HL, DE' => DE, BC' => BC), followed by an instruction that exchanges the values between these registers (to set the aux values in the aux registers, EXX) and the loading of the normal values in the normal registers (HL => HL, DE => DE, BC => BC). SP, IY, and IX are loaded right afterward as well. Finally, the interrupt mode and interrupt enable are set accordingly.

As mentioned before, in the introduction to this Section (6.9.2), .sna files push the PC to stack, unlike the .z80 file format. The PC in .z80 files is pushed to stack manually through DMA for both register-filling routines to be equal. In both cases, the routine ends with a return from the NMI (RETN). The "generate_routine()" function in

Marker	Stack Value	Example Address
SP –>	AF'	0xFF70
	AF	0xFF72
	PC	0xFF74
old SP –>	...	0xFF76

Table 6.3: The register values added to stack

"file_format_aux.c" is used to generate the register-filling routine. One of its parameters is the file type for the Stack Pointer to be updated appropriately for .z80 files since the PC has to be pushed to stack with DMA.

As mentioned in Subsection 5.6.2, the values for AF and AF' must be pushed to stack by NIOS so the routine can POP them into their registers. The Z80 uses a full descending stack, meaning the SP points to the last value that was pushed, and each subsequent PUSH decrements the SP by 2 bytes. To add values to the stack through NIOS II, the stack pointer value has to be decremented and the values themselves have to be written into memory. To add AF' and AF, it needs to decrement by 4. In the "generate_routine()" function, the SP read from the REGS struct is decremented by 4. Since the Z80 works in little-endian, it is decremented by 0x0400, to eliminate the need to flip endianness just to make an operation and flip it back. Similarly, for .z80 files, this function decrements the SP by 2 because of the added PC. This new SP value is put into the routine to load it (LD SP (sp_value-0x0400)). The data values are added to the stack by the functions responsible for fully loading the files ("load_SNA()" and "load_z80()") because they have access to the DMA interface's functions. At the end of the written routine, before its return, AF' value is POP'd into the AF register, the values are exchanged between registers (EX AF) and then the value of AF is POP'd into AF. The order of these values in the stack, along with the PC, is shown in Table 6.3.

6.9.2.2 Snapshot File Reading and Loading

For .sna files, all the data following the header is written directly into memory since the file contains its raw contents. The SD card is read in blocks, 512 bytes each, and the first block contains the header information. When the first block is read, the register values are extracted and saved in the REGS struct. After this, the rest of the data in the block is written directly to memory, starting at the address 0x4000, through the NIOS' DMA that was obtained before opening the file, after the Z80 sent a "load file from SD" command. Then the rest of the data is written to memory, block by block, until the end. Next, the register routine is generated with the values obtained from the file and written into memory.

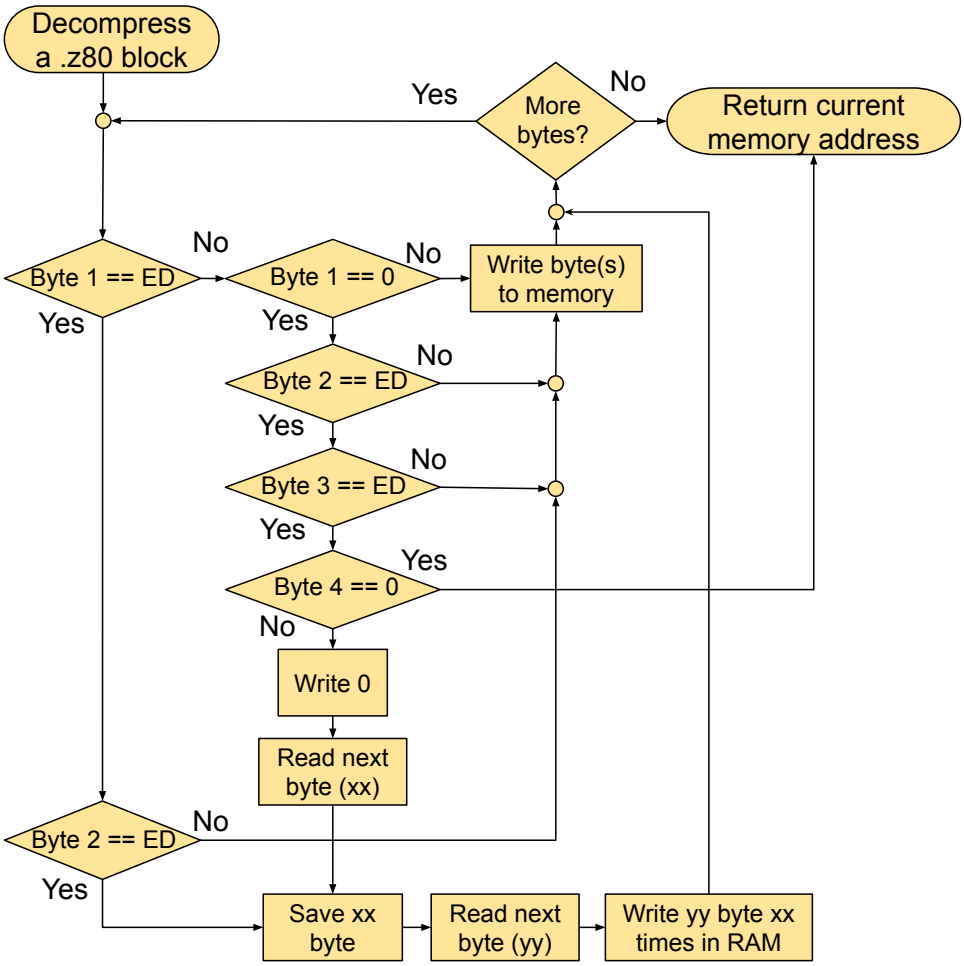

Figure 6.22: A flowchart depicting the process of decompressing .z80 files

As for .z80 files, if the file's version is 1 and it is not compressed, it is handled the same way as a .sna file. Version 1 compressed files are decompressed block by block. The decompression method for a single block can be seen in the flowchart displayed in Figure 6.22. The "Write byte(s) to memory" process implies writing any previous bytes that, in the reading process, turned out to not be actual compression tags. For instance, if 00 and ED were read (in the process of finding the ending tag) but then a different byte preceded, the bytes 00, ED, and that current byte must be written at that moment. There is no combination similar to the ending tag in version 1 files that can be found in version 2 or 3 files because a 0 following two ED bytes implies that a compressed series of bytes repeats for 0 times, which never happens.

The term "SD data block" refers to the 512-byte data blocks that are read from the SD card. ".z80 data block" refers to a compressed data block from a .z80 file. There is one for each memory page.

Reading version 2 and 3 files required more verifications based on the fact that SD

data blocks are not aligned to .z80 data blocks. For example, an SD data block could finish in the middle of a .z80 data block's header, which would require completing the header at the beginning of the next SD data block. In the first SD block, after reading the register values, the first .z80 data block's header is read, and the current writing address is set based on the page number. For every SD block, there is a check for the size of the current .z80 data block header obtained, to make sure the full .z80 data block header has been obtained. If it had not, the remaining bytes are obtained in the next SD data block and the address of where to write the data is updated. The size of the data written to memory is decided by comparing the size of the SD block with the .z80 data block's remaining data. If the .z80 data block finishes before the SD block, a new header is checked, the address is updated, and more data is written. This check is only done once per block, due to the possible minimum size of a .z80 data block, concerning a 512-byte SD block. Each compression tag can only compress up to 255 bytes, and each .z80 data block contains a compressed ZX Spectrum memory page of 16KB. Therefore, the minimum size of a .z80 block is composed of 64 compression tags compressing 255 bytes, plus an extra one for 64 bytes: $(64 \times 255 + 64 = 16,384\,bytes)$. This results in a size of 260 (corresponding to 65 four-byte tags) + 3 header bytes ($65 \times 4 = 260$). With an SD block of 512 bytes, only one full .z80 data block can fit inside an SD block, and the maximum amount of .z80 data blocks that can fit in one SD block is 2. Therefore, only one extra check for a new header is necessary. After writing all the data the register-loading routine is generated and written to memory, followed by the AF', AF, and PC values being pushed into the stack and the NMI being triggered.

The process of the main program loop for the NIOS II code, described up to this point, is illustrated in Figure 6.23.

6.9.3 SD Card Main and File Menus

The start menu and the menu used to select the file to load from the SD card were implemented based on the code for the 128k ZX Spectrum menu to maintain the appearance and mechanism.

Because of how the 128k menu's assembly code was organized, only a few data values and code had to be changed. This included the menu's position on the screen, the black outline's dimensions, the menu's contents, and the handlers of each option. Other parts of the code that required altering were calls that the 128k ROM 0 would make to the other page of ROM, ROM 1. ROM 1 is equivalent to the ZX Spectrum 48k's ROM. Therefore, these calls were altered to be simple calls to functions now accessible directly in the 48k's ROM. These details were identified when analyzing the

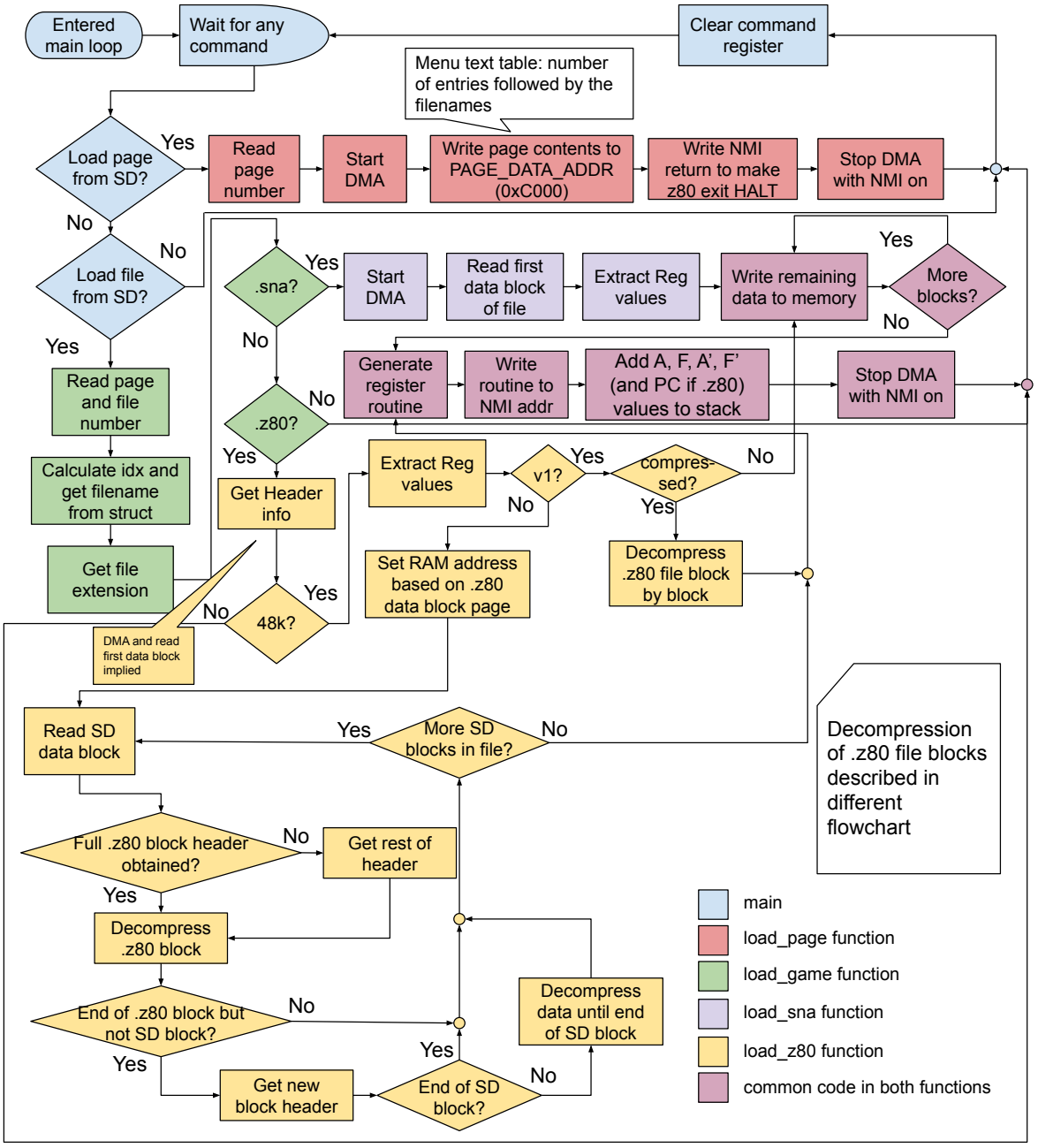

Figure 6.23: The processes involved in the main program loop of NIOS

disassembled code of the 128k's ROM 0.

For the file menu, two extra memory addresses were used to store the current page number (0xEC0B) and current page type (SD card, online optional feature (0xEC0A)).

When booting the ZX Spectrum, the new menu is displayed instead of the normal 48k basic editor. To support the use of Sinclair Joysticks to navigate the menu, entries were added to its key action table. The full key action table is displayed in Listing 6.3. The BASIC option restarts the spectrum without the main menu appearing. This was done by replacing the end of the NEW command routine with a conditional jump. By saving the state of the menu, a RESET could be triggered when the BASIC option is selected that restarts the Spectrum and, instead of jumping to the menu code, jumps to the rest of the 48k ROM code.

```
1  MENU_KEYS_ACTION_TBL:
2      DEFB $06            ; Number of entries.
3
4      DEFB $0B            ; Key code: Cursor up.
5      DEFW UP_HNDLR       ; MENU-UP handler routine.
6      DEFB $39            ; Joystick 1 UP
7      DEFW UP_HNDLR       ; MENU-UP handler routine.
8
9      DEFB $0A            ; Key code: Cursor down.
10     DEFW DN_HNDLR       ; MENU-DOWN handler routine.
11     DEFB $38            ; Joystick 1 DOWN
12     DEFW DN_HNDLR       ; MENU-DOWN handler routine.
13
14     DEFB $0D            ; Key code: Enter.
15     DEFW SEL_HNDLR      ; MENU-SELECT handler routine.
16     DEFB $30            ; Joystick 1 FIRE
17     DEFW SEL_HNDLR      ; MENU-SELECT handler routine.
```

Listing 6.3: Declaration of data variable for the menu's key action table

The initialization of the Spectrum clears all its memory when it performs a memory check from 0x4000 to 0xFFFF, so the BASIC option handler could not simply jump to address 0x0000. By analyzing the routine, it was found that the entry point used by the START command routine at the ZX Spectrum's start was different (labeled as START_NEW), for the RAM address passed as an argument was always set to 0xFFFF and therefore all memory would be checked/reset. In the normal entry point of this routine, the DE register pair is set to the value in the system variable RAMTOP, which is 0xFF57. This meant any value above the address 0xFF57 would remain, so this "menu enabled" flag was set to be in address 0xFFFF. This value is at 0 after a ZX Spectrum

reset, so the menu code is jumped to. The only time it is not 0 when passing through the verification is when it was altered by the BASIC option routine before making a jump to the start of the NEW command routine. Since the copyright message printing code was replaced with this new verification, the jump address in the verification is actually in the unused RAM region that writes the copyright message before jumping to the rest of the normal 48k ROM code (0x12A9). The verification can be seen in Listing 6.4.

```
#code EPROM, 0x1295, 0xD ; address, length

    LD  HL, $FFFF ;
    BIT 0, (HL)    ; getting bit 0 of byte at FFFF (menu flag)
    JR Z, MENU     ; if 0, go to menu code
    JP $3878       ; NORMAL_BASIC_MISSING_CODE label in unused ROM
MENU:
    JP $386e       ; MAIN label in the menu code
```

Listing 6.4: The menu verification inserted in the NEW command routine in ROM

To create the file menu, a separate series of routines was written in assembly code, without the main menu. In this assembly code, the same base menu code was used but altered to define new dimensions that would occupy the whole screen, allowing filenames with up to 32 characters, including their extensions. This file still exists in the project and is called "file_menu_only_test.asm". The menu text data is hardcoded in this file, abstracting the creation of the menu from the NIOS software that would later be used to write these file names. This menu text table starts with a title, much like the main menu, which contains the string "SD LOADER" when the SD Loader option is selected.

As was mentioned previously, the current page number as well as the current menu's type are saved in memory. Both of these serve to keep track of the address to use in NIOS commands, where the lower byte must have the type (0x1B for SD card or 0x1D for Online) and the high byte must have the page number. The page number is manipulated by the left and right key action handlers, which go and increment or decrement the page number based on the action. Checks were put in place to determine how possible these changes were, like going to the previous page if the current one was 0 or the next page if the last page was reached. NIOS stores the "remaining pages" number in address 0xEC09. It also writes a formatted string with the number of the last page requested compared to the number of pages that exist. This string is located after the array of filename strings written in memory via NIOS.

A routine called "PRINT_BLACK_STRIPE" was created that can paint full black stripes

of attribute blocks from one end of the screen to another to "book end" the menu. After determining the looks of the menu, the NIOS side was implemented and tested. The NIOS writes the text table to a determined address used by the menu code to load the options, established as address 0xC000 (PAGE_DATA_ADDR).

To load a file, when the ENTER button is detected (key press processed), a handler is run that sets up the I/O OUT command to instruct NIOS to load the file, with the page number and the game's index set up. It performs a WRITE with the menu option index as the data to address: (HIGH byte: page number)(LOW byte: interface). This code can be seen in Listing 6.5. The "OUT (C), A" instruction sets the contents of the B register in the high byte of the address bus and the contents of C on the low byte.

```
FILE_SEL_HNDLR:
    DI

    LD   HL,$EC0C
    LD   B,(HL)        ; B=Current menu option index (game number in page)
    LD   C, $1B        ; SD interface

    LD   A, ($EC0A)    ; page menu type
    AND  A             ; for checking if A is 0
    JR   Z, FILE_SEL_HNDLR_CONT
    LD   C, $1D        ; Online interface
FILE_SEL_HNDLR_CONT:
    LD   A, B          ; A=Current menu option index (game number in page)
    DEC  HL            ; HL=0xEC0B.
    LD   B, (HL)       ; B=Current menu page

    OUT (C), A
    HALT
```

Listing 6.5: The File Select Handler routine

After both menus were created separately, they needed to be combined into a single Z80 program. The routine to show the file menu is triggered by the use of the SD Loader option in the main menu. This handler makes a NIOS get page command and when NIOS is done with it, jumps to the start of its show menu routine, now called "SHOW_FILE_MENU". The command also writes the file menu's filenames to PAGE_DATA_ADDR in NIOS. The main menu's equivalent routine label is called "SHOW_MAIN_MENU". The code created for each menu combined did not fit in the unused ROM that was being used. Since some of the routines used were very similar

in both menus, these routines were altered to receive an extra parameter that determined some values inside them like pointers to text tables or handlers. These routines include:

- "PROCESS_KEY_PRESS" receives the address to the menu keys lookup table directly, so whichever menu calls it can do so with a respective address;
- "TOGGLE_MENU_HIGHLIGHT" receives a value in register C to determine where the first attribute block to highlight is. This is due to the file list menu starting farther left at coordinates (4,0), whilst the main menu starts at (12,9);
- the action handlers' auxiliary routines "MOVE_UP" and "MOVE_DN", call "TOGGLE_MENU_HIGHLIGHT" with the C register prepared (0 for main menu).

These changes to the routines allowed the code to fit within the unused 48k ROM space with no room to spare. A flowchart describing what the SD Loader option does can be seen in Figure 6.24. The process begins by halting the CPU. Then NIOS writes the filenames of the first file page and the menu is loaded with that data. When the user presses the up or down arrows, the index of the menu option is incremented or decremented and the highlight graphic is updated. When the user presses the left or right arrows, the page index is updated and a command is sent to NIOS with the new page index. NIOS stores the filenames in PAGE_DATA_ADDR and then the menu is loaded again. When the user presses SEL (ENTER), the Z80 sends a command for NIOS to load a file. It sends this command with the page number and the option index, which is then used by NIOS to obtain the original filename. With this, NIOS can read the selected file and load it, as described in Subsection 6.9.2.

6.9.4 Save State Functionality

Save state functionality required the use of an SD/FAT library with write support. The DE2-115 SystemCD's SD card demonstration lacked write operations, required for creating and writing the snapshot files generated. The library "FatFS - Generic FAT Filesystem Module" was used. It includes many options to scale the size of the library based on the requirements of the user. It required the implementation of low-level device controls to communicate with the SD card. These were implemented based on the SystemCD's SD card demo code.

The process of saving the machine's state is started by pressing a push button on the target platform, specifically KEY 3. This sends a command to NIOS (command 0x19).

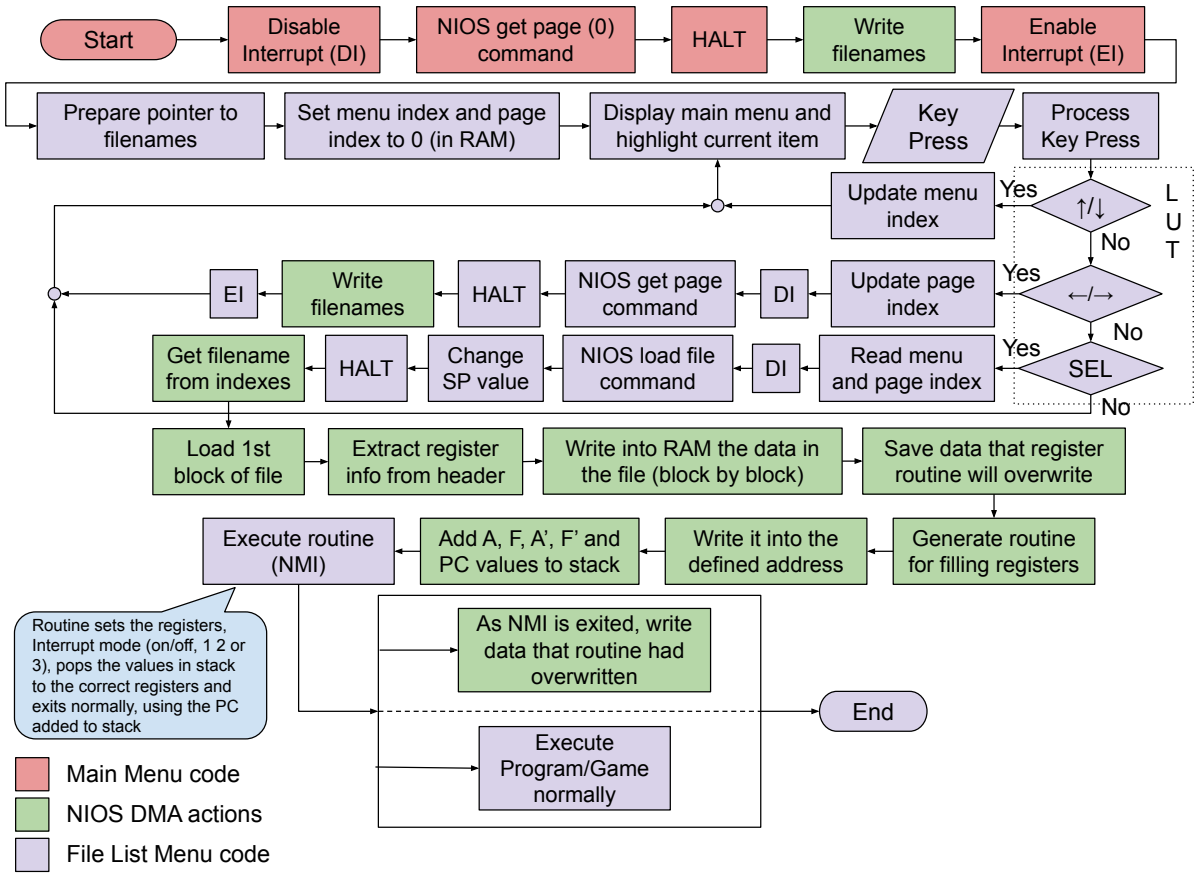

Figure 6.24: A flowchart of the file menu's processes after the SD Loader option is selected

Marker	Stack Value	Example Address
	AF'	0xFF70
	AF	0xFF72
SP –>	PC	0xFF74
old SP –>	...	0xFF76

Table 6.4: Stack after the save registers routine

NIOS then requests DMA, stopping the Z80, and saves the first 485 bytes from video memory locally in a buffer. This data will be present in the first sector of the file to be created, and the 27 bytes before it will correspond to its header, composed of register values and interrupt/border information. NIOS then writes a routine that saves the contents of the Z80's registers to memory. The save routine is written to screen memory and, as it is being executed the register values are also being written in that region. This overwrites the routine as it is being executed, saving space. As previously mentioned, AF' and AF don't have set/get instructions, so they are pushed to stack after the SP is saved. The SP is therefore pointing to the PC, which is required for .SNA files, but obtaining AF' and AF is still possible via NIOS by decreasing the SP value by 4, as seen in Table 6.4. Triggering the NMI pushes the PC to stack initially, so the old SP in this table represents its state before the NMI.

NIOS triggers an NMI as it stops DMA and the Z80 executes the routine. At the end of this routine, another command is issued to NIOS (0x19 read) to notify it of the availability of the register values. NIOS begins a new DMA, extracts these values, and writes them to the locally stored first-sector buffer. The file is created with a number appended to the original software's name if one was loaded or simply named "save_xx.sna" (numbers where xx is) if no software was previously loaded. Files exceeding 32 characters do not show up in the list, so the created files might not appear in the menu unless their names are changed elsewhere. The first sector is written to the file and the proceeding sectors are read from memory, 512 bytes at a time, and written to the file.

The routine that saves the register's values to memory misplaces the register's contents in the process. The routine for loading a file's registers into the Z80 is used to load the values back, therefore resuming the software after the operation. Some header data is not saved, however, since there are no Z80 instructions to obtain them. This includes the interrupt mode (0, 1, or 2) and the interrupt flip flops inside the Z80, which determine the state of the interrupt (on/off). Default values are used for both.

6.10 Corrections to the Proposed Architecture

Some games that were loaded would have issues in executing through SD Loader but worked well when loading through audio tape. Some games would cause a ZX Spectrum reset as soon as NIOS finished loading them. Others would do this at other times during their execution. There was no easy way to test these problems since it was unknown how to run ModelSim, the testing program, with VHDL files + NIOS instance + software running in NIOS.

At one point, a possible situation was identified, one that had not been considered yet: the NMI causing a write to memory when it would trigger, specifically the return address being added to the old SP before NIOS starts writing to memory. This meant that, somewhere in the game's code, two bytes were being overwritten and causing problems. A simple bug fix was thought up, based on the stack being descending, as well as the Z80 having a "LD SP" instruction. So, before the instruction to load the game is sent to NIOS, the SP is changed to 0x57FF, the last address of screen data. This means that the address before the NMI is executed is instead pushed to the bottom right of the screen to avoid problems with code being overwritten. This fixed some of the games, such as Chase HQ and Bubble Bobble.

Bomb Jack still had a problem running correctly, through SD Loader as well as when loaded through audio tape. To make sure that the problem was not being caused by any of the ROM alterations that were made, the 48k ROM was restored and the game was loaded through audio input with the LOAD "" command. In this situation, the game loaded correctly, so the conclusion was that some ROM alteration caused this. Perhaps something like the game using the unused ROM space to obtain the value 0xFF quicker? It is unknown. So, the solution was changed to clear up as much space in the unused area of ROM by making the menu code instead be written by NIOS at the start of the ZX Spectrum.

This time, the assembly code that decides if the menu is to be loaded was moved to the unused ROM space. In this region, it would occupy significantly less space than the full menus. The only alteration in the NEW command's routine is a jump to this code, instead of replacing multiple instructions in it. In case no menu is necessary (BASIC option selected), the code can be in the unused region to avoid NIOS interaction in verifying the menu flag.

Every time the ZX Spectrum initializes, RAM is checked and reset. This means NIOS can't write the menu to memory on boot. Therefore, a new command for the peripheral interface was created called "INIT". The Z80 sends this command when it executes the

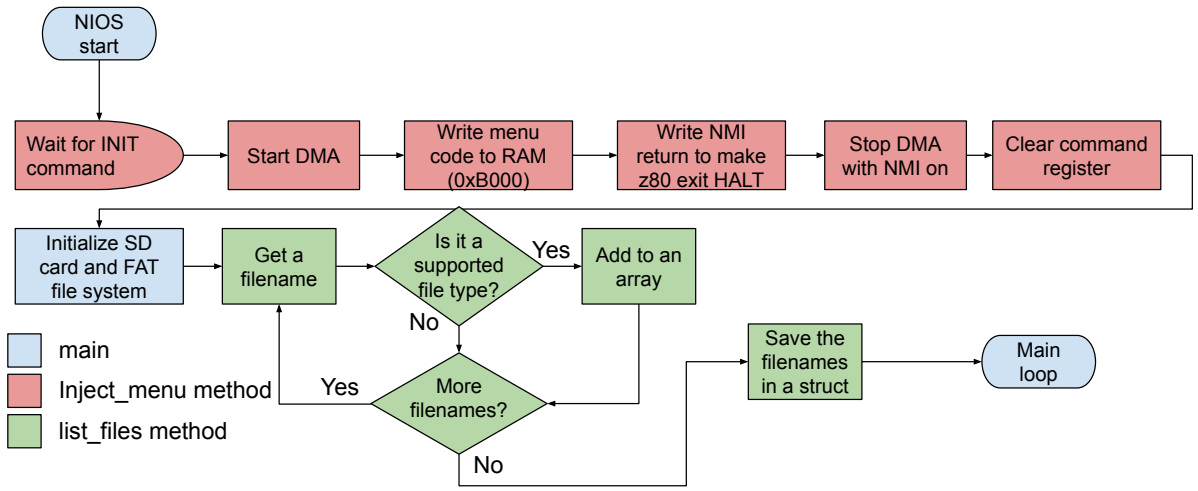

Figure 6.25: A flowchart of NIOS's initialization

routine in the unused ROM region when the main menu is required. Therefore, when the BASIC option is selected no menu is written after the system reset. After NIOS writes the menu code, it triggers an NMI for the Z80 jump to that code. The menu code address is defined as 0xB000 with the name "MENU_CODE_ADDR" in "main.c".

Since the Z80 waits for the NIOS to finish writing the menu to memory, if the software is not running on NIOS, the screen will simply have no options listed on it. To return from the Z80's HALT state, a simple return routine is written in the defined NMI routine address (0x4000, top left video screen data), and the NMI is triggered, leading to a flickering glitch on the screen with each page change, which is then overwritten to clear it again.

A flowchart showing NIOS' process when it initializes can be seen in Figure 6.25. It starts by waiting for the Z80's INIT command, to make sure the initialization process of the Z80 is finished. Next, the menu code is written to memory and an NMI is triggered to remove the Z80 from its HALTed state. Then, all the filenames of the supported files in the SD Card are saved. Figure 6.26 consists of a flowchart of the ZX Spectrum's initialization, as well as the main menu. When the ZX Spectrum is activated, the Z80 executes the initialization of the ZX Spectrum as normal but then verifies the menu flag (address 0xFFFF in RAM) to know if the menu requires loading. If so, NIOS writes the menu code to RAM and it is executed. This code creates the main menu, updates the highlighted option for every up or down input received, and executes a routine for the option selected, as was previously mentioned.

A diagram of how the ZX Spectrum's ROM modifications are organized, after all these bug fixes, can be seen in Figure 6.27. Bomb Jack is now playable on the ZX Spectrum Fusion.

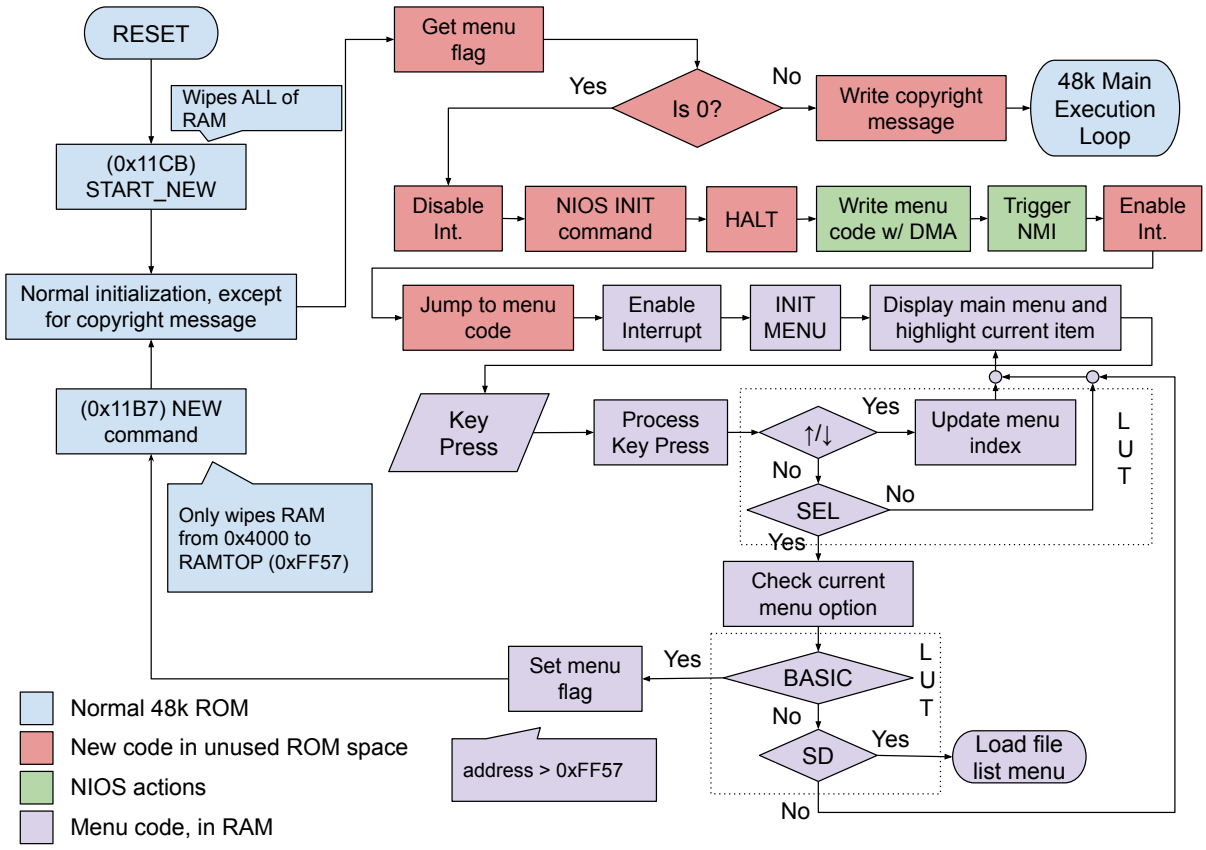

Figure 6.26: A flowchart of the ZX Spectrum Fusion's initialization

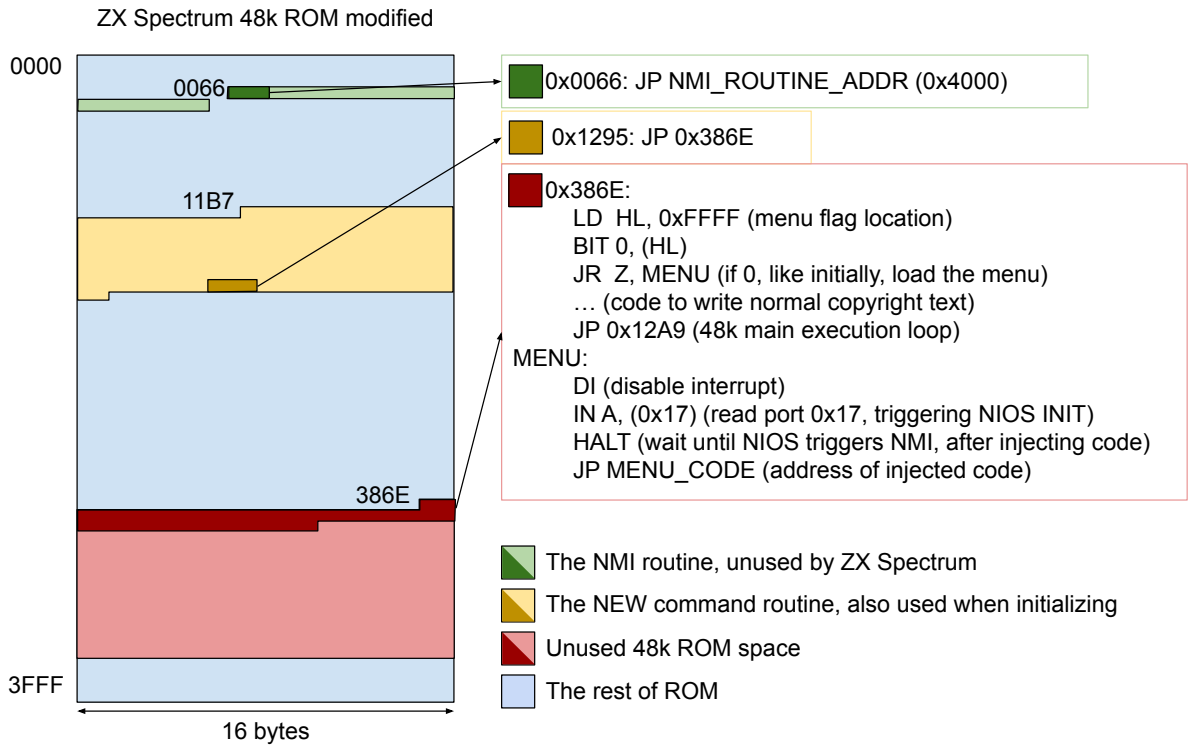

Figure 6.27: A diagram of the ZX Spectrum 48k ROM's modifications

The on-chip memory spent on the filenames by NIOS was reduced by saving only the filenames that can be shown on screen at a time, unlike in Figure 6.25. When a page is requested, only the 16 filenames of that page are saved. Figure 6.28 shows a flowchart of this process which is executed by calling the function "list_files_of_page()" in sd_if. The file entries are iterated until the first file of the requested page is reached. Only supported files are counted in both loops. The second loop stores up to 16 filenames of supported files.

If no limits were to be set on saving the filenames, the necessary memory to store all filenames of a large SD card would not be sufficient, i.e. a 16GB SD Card can store 262,144 snapshot files, resulting in 8,388,608 bytes of on-chip memory, which is not available.

6.11 FPGA Resource Utilization and Timming

The synthesis tool (Quartus) reports timing and resource information after a design is compiled. This information is used to conclude how many resources are occupied on the device and what its maximum performance is. The following resources were used:

- Maximum clock frequency: 69.76 MHz;

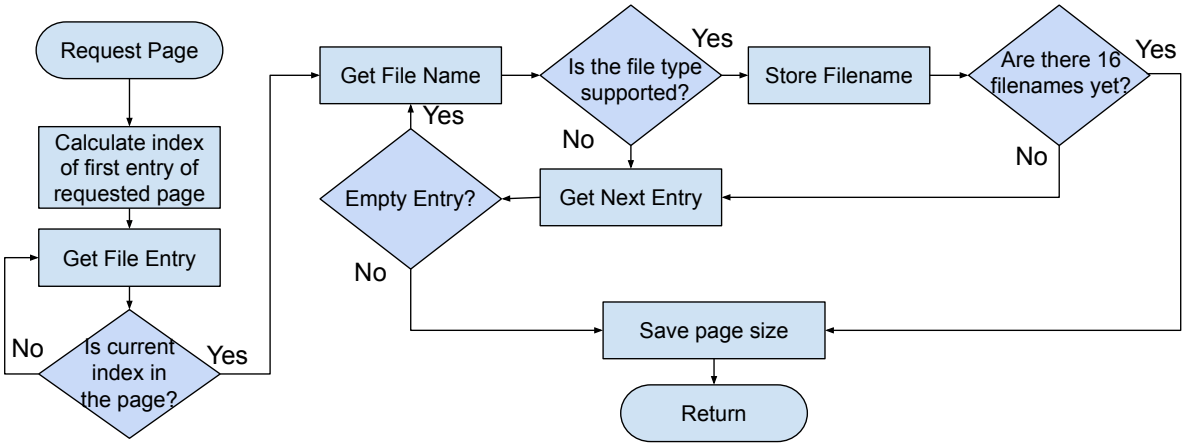

Figure 6.28: A flowchart of the "storing filenames of a page" function

- Total LEs: 6,463/114,480 (6%);
- Total memory bits: 1,322,016/3,981,312 (33%);

Figure 6.29 shows a diagram of the FPGA chip and how much of it was used for this implementation.

Only one PLL was used and no DSP blocks were used in this design. The resource utilization per entity can be seen in Table 6.5. This table shows that the "T80a" implementation of the Z80 uses the most logic cells out of all other components. This is worth comparing to the new processor introduced ("nios_sd_loader_cpu"), used in the "nios_sd_loader" since it could be considered an overhead that is too big for the overall system. Nevertheless, the resource utilization results show that the NIOS component is using the most memory out of the rest. Inside "nios_sd_loader", the on-chip memory is occupying 262 M9Ks, that is, 2,146,304 memory bits. The software's size in the compiled ELF binary was 205 KB for code + initialized data, and 43 KB were free for stack + heap. So, the total memory necessary to hold the program is $205\,KB \times 1024 \times 8 = 1,679,360 bits$. The NIOS program in this context does not require this much free space on stack or heap. Now that the program has been profiled, its memory footprint can be reduced. It is considered that 1 KB is enough space for stack in this context. The size of the on-chip memory was initially set to make space for the implementation during development. Without any other optimization or modification other than the stack/heap size, the total required space would be 1,687,552 bits. So, only 206 M9Ks ($1,687,552 bits/8,192$) would be necessary to implement the system, leading to an approximate reduction of 21.37% when compared to the 262 M9Ks currently used by the NIOS on-chip memory.

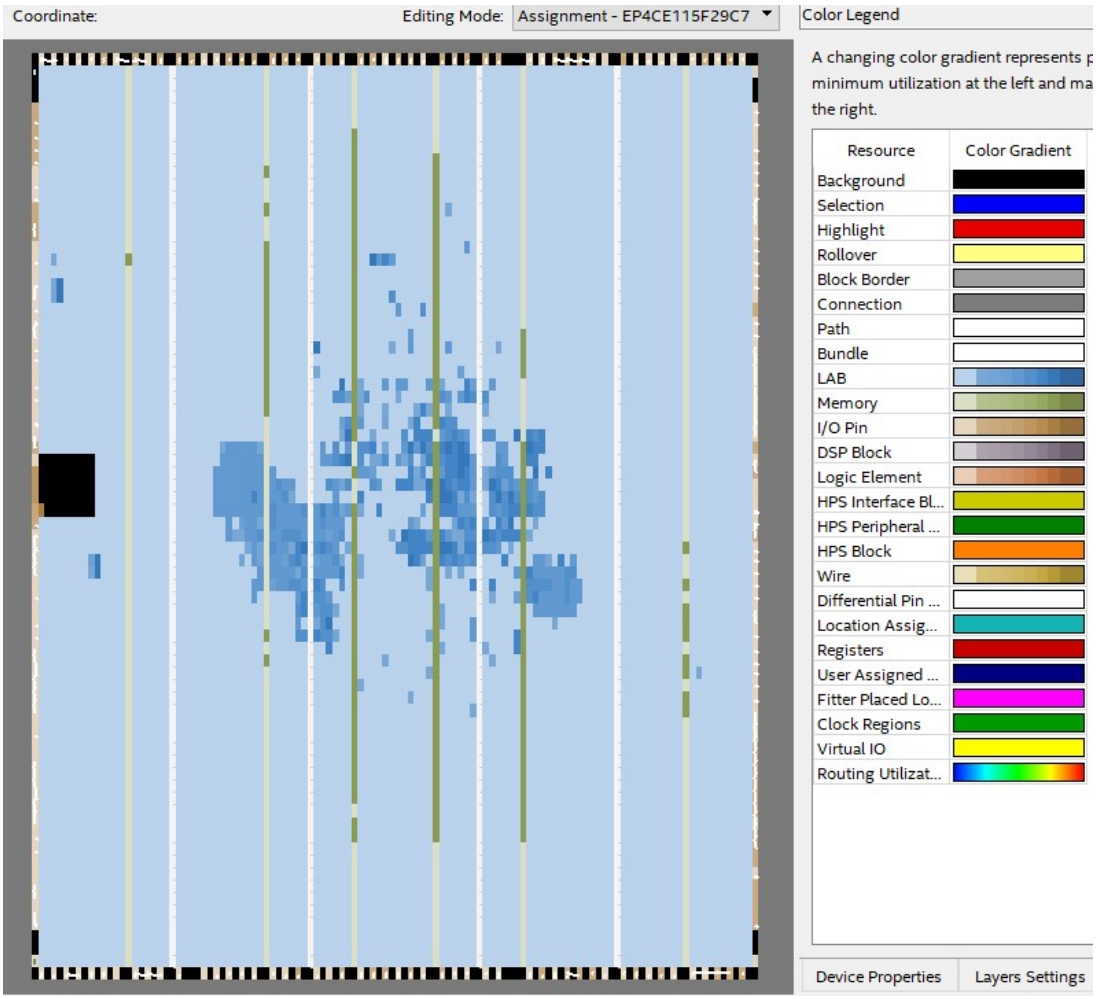

Figure 6.29: A view of the FPGA chip showing the utilized blocks. Dark green squares represent memory units and dark blue squares represent LABs (1 LAB = 16 LEs)

Node	Logic Cells	Registers	Memory Bits	M9Ks
top	6,543 (228)	2190 (0)	132,1976	168
T80a	2,498 (24)	360 (15)	0	0
audio_adc	8 (8)	6 (6)	0	0
audio_codec	88 (88)	66 (66)	0	0
color_video_ram	0 (0)	0 (0)	6,144	1
kempston_if	9 (9)	0 (0)	0	0
keyboard_top	793 (0)	181 (0)	0	0
nes_gamepad	62 (62)	42 (42)	0	0
nios_per_reg	26 (26)	0 (0)	0	0
nios_sd_loader	2,222 (0)	1,216 (0)	2,108,424	262
pixel_video_ram	0 (0)	0 (0)	49152	8
pll	0 (0)	0 (0)	0	0
remaining_ram	147 (0)	54 (0)	337,912	42
reset_counter	10 (0)	10 (0)	0	0
rom	79 (0)	46 (0)	131,072	16
ula_count	46 (14)	23 (0)	0	0
ula_port	11 (11)	10 (10)	0	0
video	171 (27)	49 (0)	0	0

Table 6.5: A table of the resource utilization of each component

Regarding the sizes of the ROM ($131,072\,bits = 16\,KB$), the Remaining RAM, the Color Video RAM, and the Pixel Video RAM ($337,912 + 6,144 + 49,152\,bits = 393,208bits = 48\,KB$), it is verified that their sizes correspond to the ZX Spectrum 48k computer system.

6.12 Summary

All the main components have been implemented successfully. The most complex part of this implementation is the NIOS and its interactions with the T80, the memory and I/O, and with the SD card, as well as the file reading and decompressing. These components were not all developed at the same time, however, since they required testing to make sure they worked before adding complexity to the project. The chapter that follows is dedicated to this process.

7

ZX Fusion Evaluation

This chapter presents the evaluation of the proposed novel ZX Spectrum for correct operation and compatibility with the original computer system. The evaluation of this implementation included simulations and unity tests for all hardware components, debugging of programs for the Z80 in Assembly and Basic, and for the NIOS II in C code.

In addition, incremental integration tests were made to validate the inclusion of each component on the system as they were implemented.

7.1 Video

The video was one of the first components to be tested. It was tested by outputting screen data to the VGA interface that was loaded into a memory module, which was obtained from a ZX Spectrum emulator running the game Manic Miner. This avoided having the rest of the system up and running since the video component only needed the memory to read and display appropriately.

A test-bench was also created for this video component. It shows that the video data was delayed by 1 pixel, but since this effect is consistent throughout the screen, it is not noticeable. The diagram in Figure 7.1 shows the address signals updating when $x = 0x180$, which is the first pixel of the next attribute block. Half a clock cycle later, the data is loaded from memory and half a clock after this, the data is displayed.

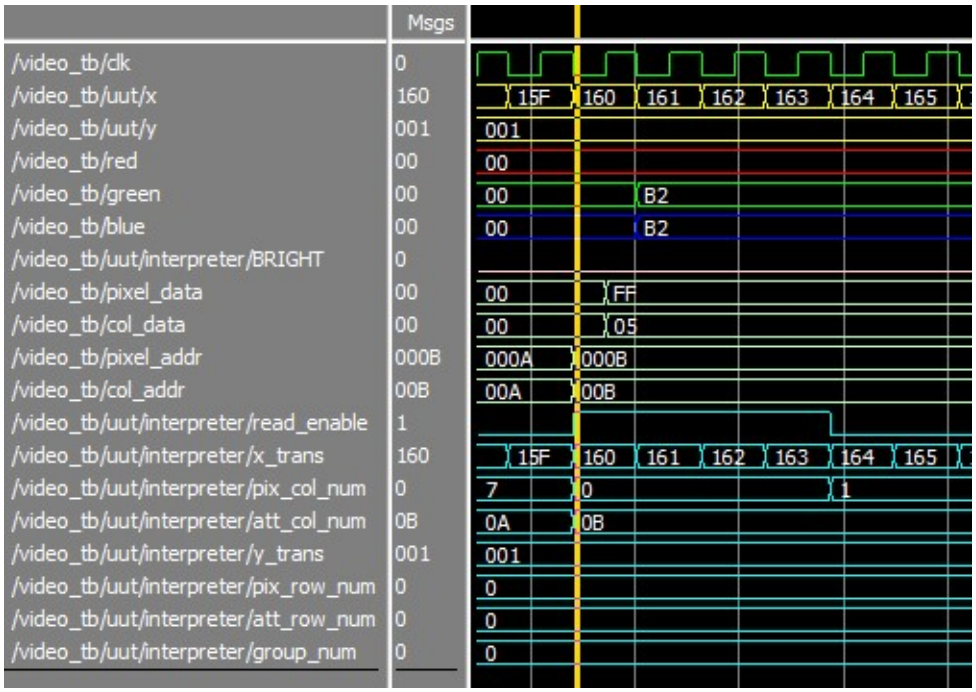

Figure 7.1: The video component's signals when a new attribute block is beginning to render

The border works as the original screen border. When tapes are loaded or saved, the border color is changed between two colors rapidly to make a multicolored border effect. This implementation's border is able to have the same effect during these operations. However, some ZX Spectrum programs draw in the border by changing its colors at specific timings, based on the television video frequency (50Hz). Known examples are "Aquaplane" which stretches the sky to water effect into the border itself and "Dark Star" which draws what appears to be a ship pointing upwards above the results screen. These effects are not supported in this implementation of the border drawing, in none of the different resolution options, since it is rendered in 60Hz (with a pixel clock of 65MHz), which is faster than the analog TV, making the border change timings of the original software different. It also has a different size to the original in all of the resolution modes. The three resolution modes, 4x, 2x, and 1x, can be seen in Figure 7.2, Figure 7.3, and Figure 7.4, with a red border, respectively, with a red border in order to better highlight the screen's size.

7.2 CPU and Memory

The T80 was initially tested in a standalone project to check if it was working correctly. This project mapped the clock signal to a push button on the FPGA board and the data

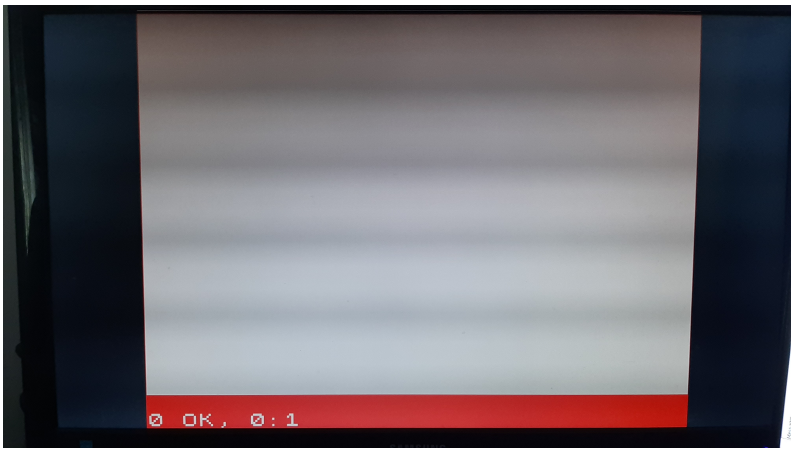

Figure 7.2: The full resolution, with a 1-pixel wide border, in 1024x768

Figure 7.3: The intermediate resolution, 2x resolution of the ZX Spectrum (512x384) in 1024x768

Figure 7.4: The smallest resolution, 1x resolution of the ZX Spectrum (256x192) in 1024x768

bus to switches, for a manual step-by-step execution. It displayed the control signals in LEDs and the address and data busses on 7-segment displays. A simple assembly code was loaded manually via the switches to test some of its instructions, specifically a read, an addition (modify), and a write. Since this was done to test the T80 on its own, the data input acts as a makeshift memory, where the data is set when the CPU is reading, and the result of the addition can be seen in the data output during the store instruction. The instructions themselves were input as their respective opcodes when the CPU would read the PC address. The assembly code used for this test can be seen in Listing 7.1. The instruction machine codes inserted in the data bus are in parentheses in the comments in front of each instruction.

```
1 #code EPROM, 0x0000, 0x7 ; address, size
2
3     LD  HL, $4000   ; (21 00 40) Loads 0x4000 to registers H and L
4     LD  A, (HL)     ; (7E) Loads contents of memory in 0x4000 to A
5     ADD A, 1        ; (C6 01) Adds 1 to the contents of A
6     LD  (HL), A     ; (77) Stores contents of A in address 0x4000
```

Listing 7.1: T80 assembly test code to read, modify and store data

To test the memory sub-system connected to the T80, a simple program was loaded into the ROM that writes to screen and color data addresses (above 0x4000) to draw a magenta and cyan flashing X on the top left of the screen. The video component was also present in this project, which allowed for visual confirmation that a "magenta and cyan flashing x" was displayed. After loading that program into the ROM and turning the system on, the X could be seen, on top of the Manic Miner screen, as shown in Figure 7.5.

After this small ROM test, an original ZX Spectrum 48k ROM from [57], and loaded into the ROM component. It made the screen display the copyright message upon the system's boot, as seen in Figure 7.6.

7.3 Keyboard

The PS/2 keyboard component was implemented and tested on a stand-alone hardware project, where the CPU address (keyboard half-row multiplexer) was set using the FPGA board switches and its LEDs. The LEDs would light up depending on the rows selected with the input address and the keys being pressed. The keyboard was working correctly so it was added to the project with the T80 and video component,

Figure 7.5: The X drawn by the CPU's execution of instructions, circled in blue (this image still has the glitch mentioned prior, but what matters in it is the X drawn in the corner)

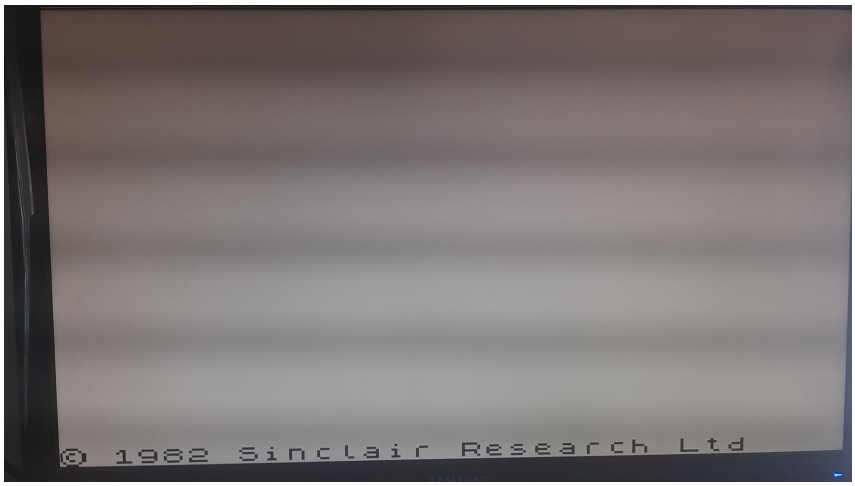

Figure 7.6: Copyright message on the boot screen of a 48k model showed after storing the original ROM.

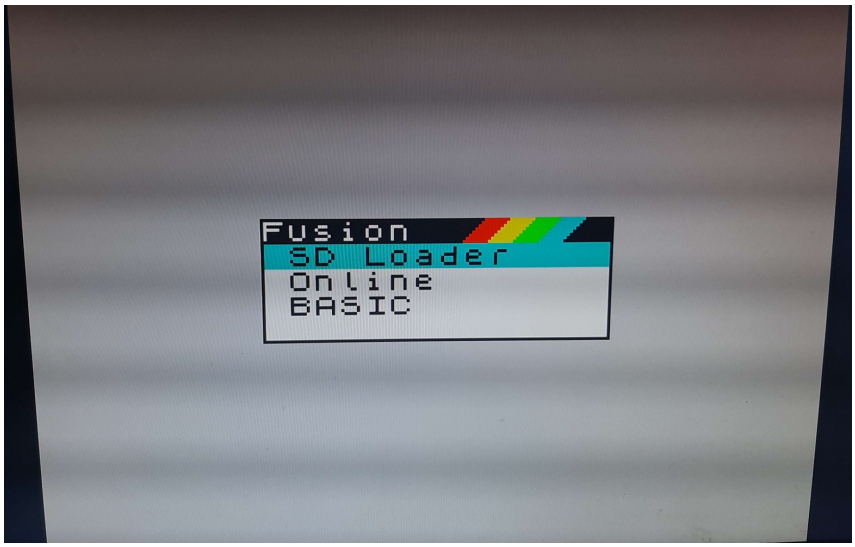

Figure 7.7: The main menu screen

along with the ULA IP Block. The ULA had to be added since it is responsible for triggering the interrupts that scan the keyboard.

An original ZX Spectrum+ keyboard was connected to the expansion board to interface with the new system. The connections consist of the data and address lines. The ULA's keyboard data bus is connected to the keyboard's output and the CPU's address is connected to its address input.

In the diagnostics ROM, which is later detailed in Subsection 7.7, one of the screens allows testing the keyboard, key-by-key. A look at the keyboard's data in the standalone project reveals the corresponding LEDs for each key.

7.4 SD Card

The SD card reading was tested by loading files of different formats, including versions, and of different lengths. Moreover, the menu navigation was tested for the file selection.

When booting the ZX Spectrum, the menu shown in Figure 7.7 is displayed instead of the normal 48k basic editor. The title "Fusion" was chosen as this Spectrum implementation was named "ZX Spectrum Fusion".

An SD card was prepared with a FAT32 file system. multiple files, with different filename lengths, were tested. Page 2 of 3 from an SD card with multiple files can be seen in Figure 7.8. The top file is highlighted and moving the up and down arrows highlights the respective files.

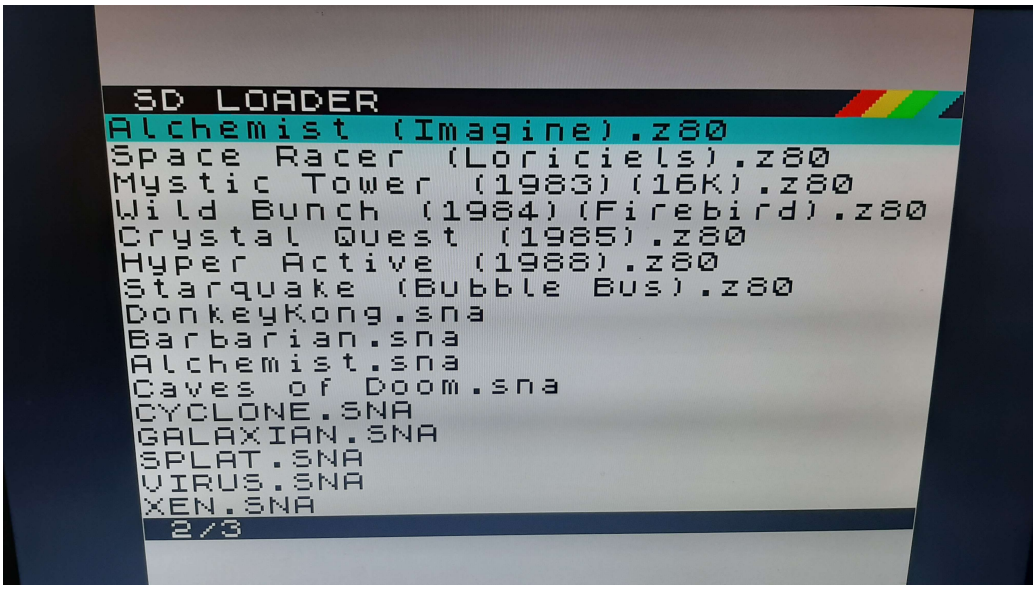

Figure 7.8: SD Loader listing the files on the SD Card, page 2/3

The files were loaded to check if they ran as expected. Bomb Jack II is seen running in this implementation in Figure 7.9.

The save state functionality was tested in some games. The resulting file is created successfully after the save state button is pressed. An example of these resulting files can be seen in Figure 7.10. It shows two save states for Bubble Bobble and one for CYCLONE. The games are also able to resume afterward. When the file is loaded from the file menu, the software loads in the state that the save state was requested in. However, problems could arise if the interrupt mode and the Interrupt Flip-Flop's states matter in the instance that the save state is created. These values are not saved to the file based on the actual state of the machine because there is no direct way of obtaining them through Z80 assembly code.

7.5 Audio

The audio was tested by hearing the sound output generated. To check if the sound was output correctly, it was compared against online recordings of the same games. It was audibly the same.

Audio input was tested by using an Android phone application called "ZX Tape Player" to play audio tape files, connected to the audio input on the target platform. The ZX Spectrum successfully loaded multiple tape files.

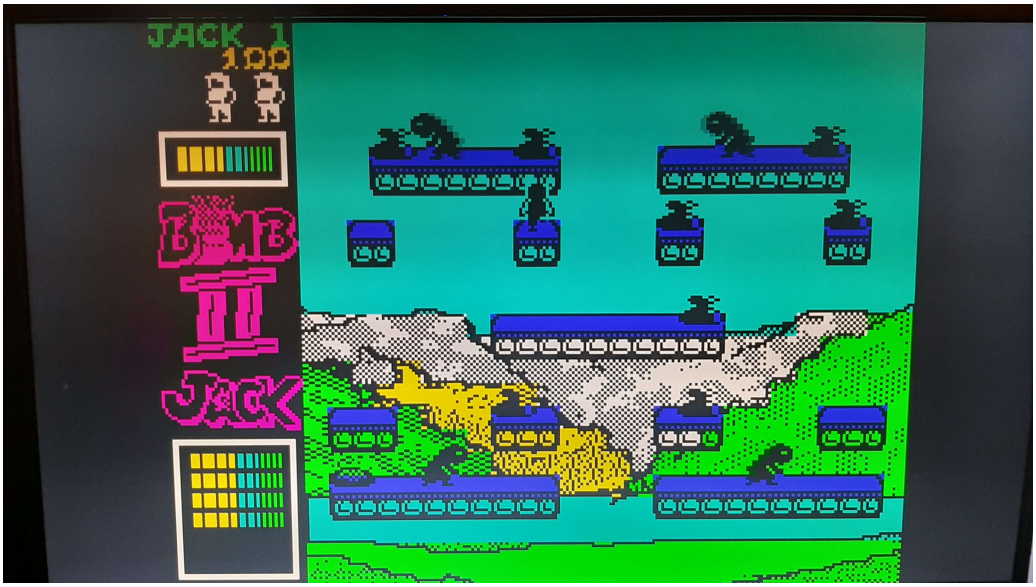

Figure 7.9: Bomb Jack II in-game

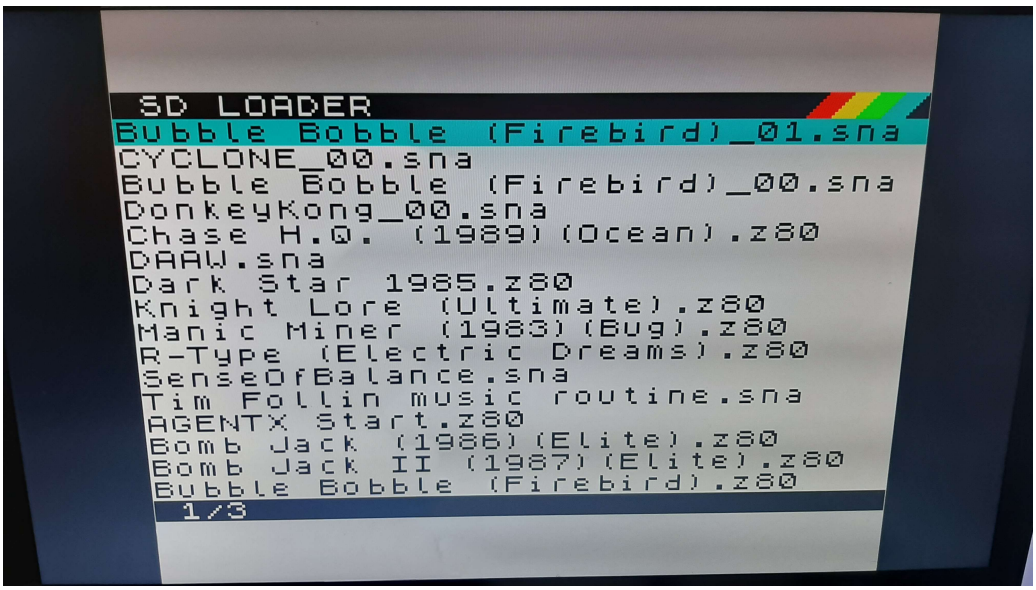

Figure 7.10: The file menu with generated save states

7.6 Sinclair and Kempston Joysticks

The NES gamepads were tested separately in a hardware project titled "nes_gamepad_test". In it, the data being received was displayed in the board's LEDs, where it was verified to be correct.

The Sinclair joystick interface was tested on the BASIC prompt since these mapped to the numbers of the keyboard. As buttons on the controllers were pressed, the correct numbers were observed.

The Kempston joystick interface was tested with a small program written in BASIC. This program reads the 0x1F port of the Kempston interface and outputs a letter (R,L,D,U,F) based on the value being read corresponding to the pressed button. This code can be seen in Listing 7.2.

```
1 LET x=IN 31
2 PRINT AT 0,0
3 IF x=0 THEN PRINT " "
4 IF x=1 THEN PRINT "R"
5 IF x=2 THEN PRINT "L"
6 IF x=4 THEN PRINT "D"
7 IF x=8 THEN PRINT "U"
8 IF x=16 THEN PRINT "F"
9 GO TO 5
```

Listing 7.2: A BASIC program used to test the Kempston joystick

7.7 Diagnostics ROM

A ROM used to diagnose ZX Spectrum hardware was used to evaluate some of the features based on how they were identified by the tests. Also, it provides a simple interface to test features like the keyboard and sound I/O. The project associated with the ROM is called "zx-diagnostics" by Rendanalford on GitHub. [49]

The testing ROM has multiple diagnostics screens:

- a memory test, exercising lower and upper RAM, as well as a soak test, repeating the memory test multiple times (Figure 7.11);
- a keyboard test, which shows what keys are being pressed (all of them worked);

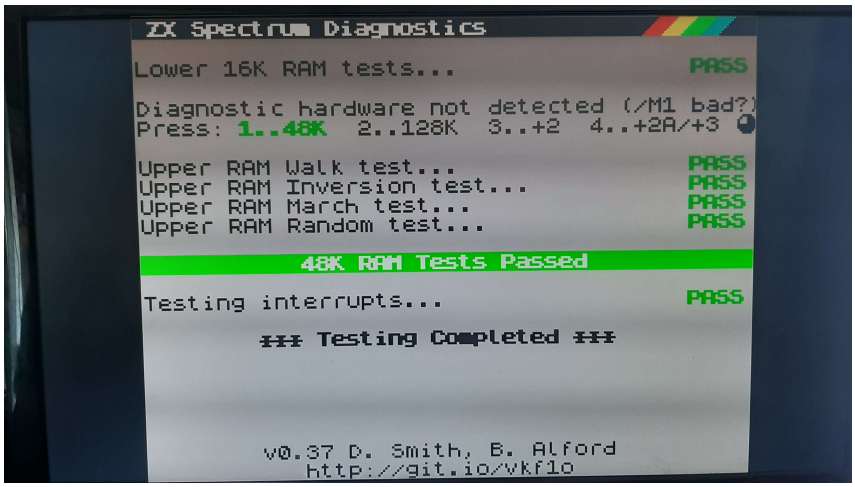

Figure 7.11: RAM test result

Figure 7.12: Visual test result

- a color test card, showing the multiple combinations of background and foreground colors the ZX Spectrum can output, as well as a repeating audio tone. The results can be seen in Figure 7.12 and can be compared to Figure 7.13, an image provided by the creator of the diagnostics tests. The beeps that can be heard during the test can be visualized in Figure 7.14;

- an ULA test, which displays the identified ULA type, the CPU type, the data present when reading the ULA's port, an interrupt test based on moving attribute blocks (the words "FAIL FAIL FAIL" appear if interrupts are failing), the options to output sound to MIC and EAR ports, generate a border with all the spectrum's colors and an ULA port addressing test (tests port 0xFE and 0xFF to make sure the ULA only responds when the write port is written to, showing the border flash green)

The ULA test, as seen in Figure 7.16, detected an ULA type of issue 1, despite the

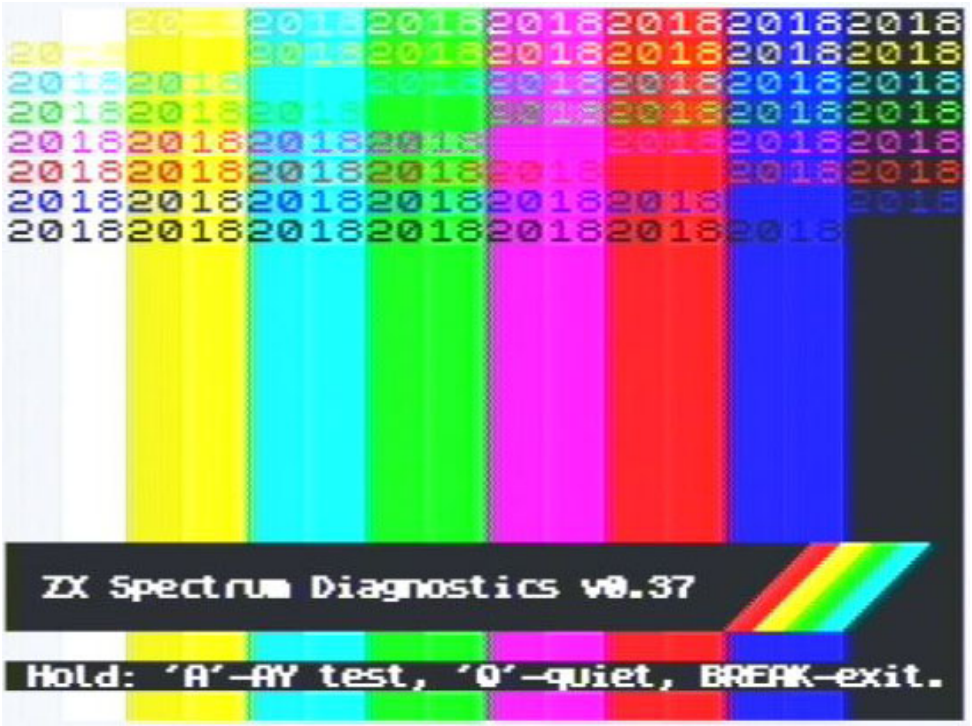

Figure 7.13: Original ZX Spectrum visual test result

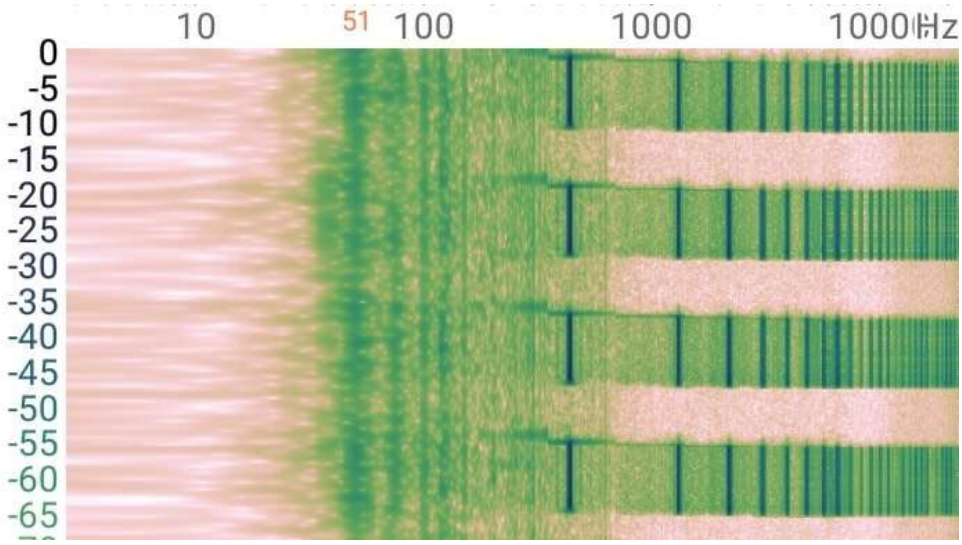

Figure 7.14: Spectogram of the visual card screen's beeping

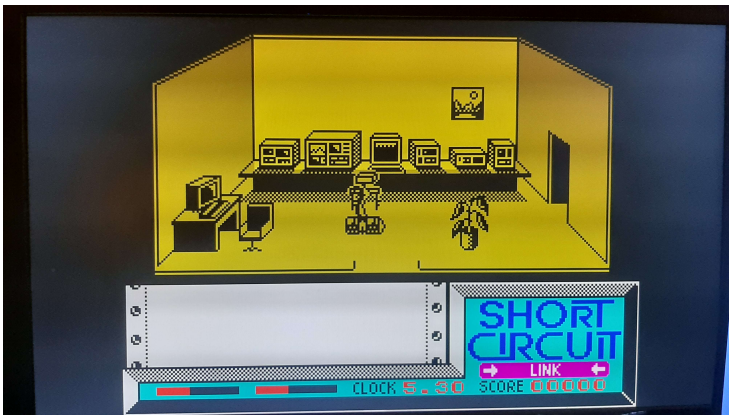

Figure 7.15: Short Circuit in-game

implementation being an issue 3. This is because the floating bus effect was not implemented, and this was interpreted incorrectly by the test. To simulate the floating bus effect, the data being read by the video component is sent to the CPU's data bus when a read is made on an unused address. With this, the test ROM was able to detect data in the floating bus, meaning games that used it now work. However, since the video fetching is at a different frequency from the original ZX Spectrum, whatever data the game obtains from the read will not be what it expects from a normal ZX Spectrum. One of four known games that have this check, Short Circuit by Ocean, was tested. Without the floating bus simulation, it would not pass the game's menu, since it would be stuck in an infinite loop waiting for data to appear in the I/O port 0xFF. With the floating bus simulated, it loaded in-game and was controllable. The game was compared to recordings of the original game, found online, and it was determined to be playing correctly. Figure 7.15 shows the game running in the implementation.

Moreover, the ULA type detected in this test screen was SLAM48+, as seen in Figure 7.17. No information could be found exactly on this designation. It is speculated that this could be related to the enhanced implementations of the ULA known as ULA plus. It is uncertain why the test detects this ULA type, but it could be because the timings used by that type of ULA were coincidentally recreated in this implementation. The CPU type is reported as CMOS (low-powered implementation, not the original), but this is not consistent, as sometimes, after programming the board again, the CPU type can be seen as NMOS, shown in Figure 7.18.

Despite these differences, games were checked to make sure they would still operate as intended. [15]

In this same test screen, the border is filled with alternating black and red lines. This border effect varies depending on the level of the audio input. A sound test was used, which swept through multiple frequencies, from low to high, to observe the effect it

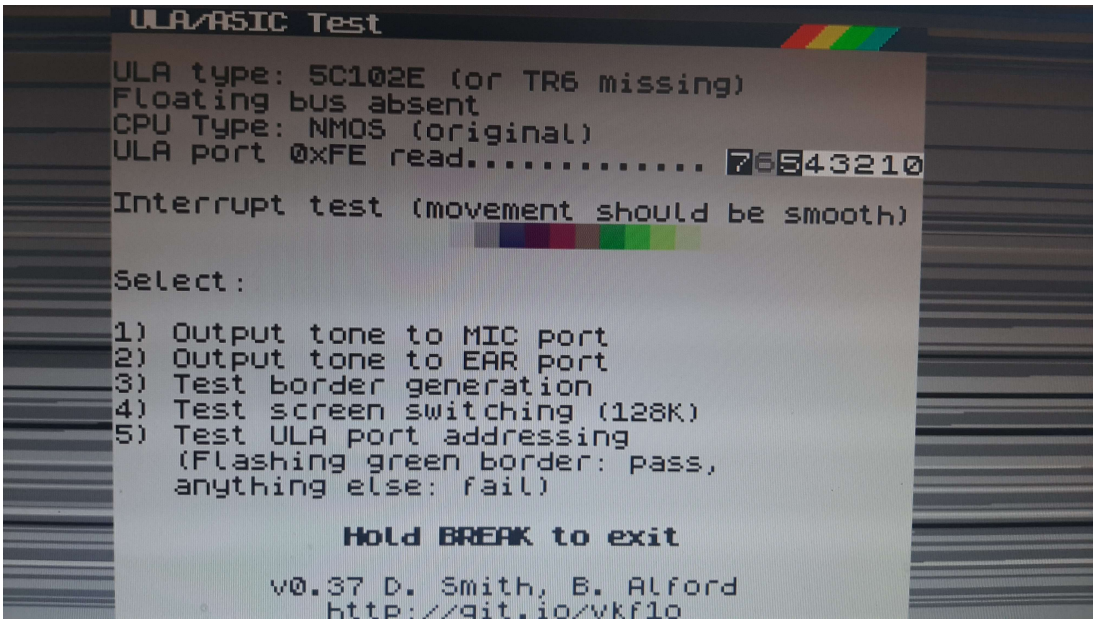

Figure 7.16: ULA test result

would have on the border. The lines in the border grew thicker with lower frequencies, and thinner with higher frequencies. This border effect was implemented in the test originally to test if the tapes were outputting data correctly, suggesting the angle of the tape deck be adjusted for more reliable loading. [50]

The option to output a tone to the EAR port was tested and can be seen working in Figure 7.19. The border generation, which creates a rainbow effect (with the spectrum's 15 possible colors) in the border, also worked, although probably with a different visibility to the original due to the border differences detailed in 6.3.3. The ULA port addressing test also succeeded, as the border only flashed green whilst holding '5'. The interrupt test, performed on this test screen, ran for several minutes without any problems.

7.8 Summary

All things considered, most of the software made for the 48k ZX Spectrum should be able to run correctly in this new system, except for games using the border to draw graphics and perhaps even those rare games that used the floating bus trick (since the data received by the CPU is not coherent with how it would be in original hardware).

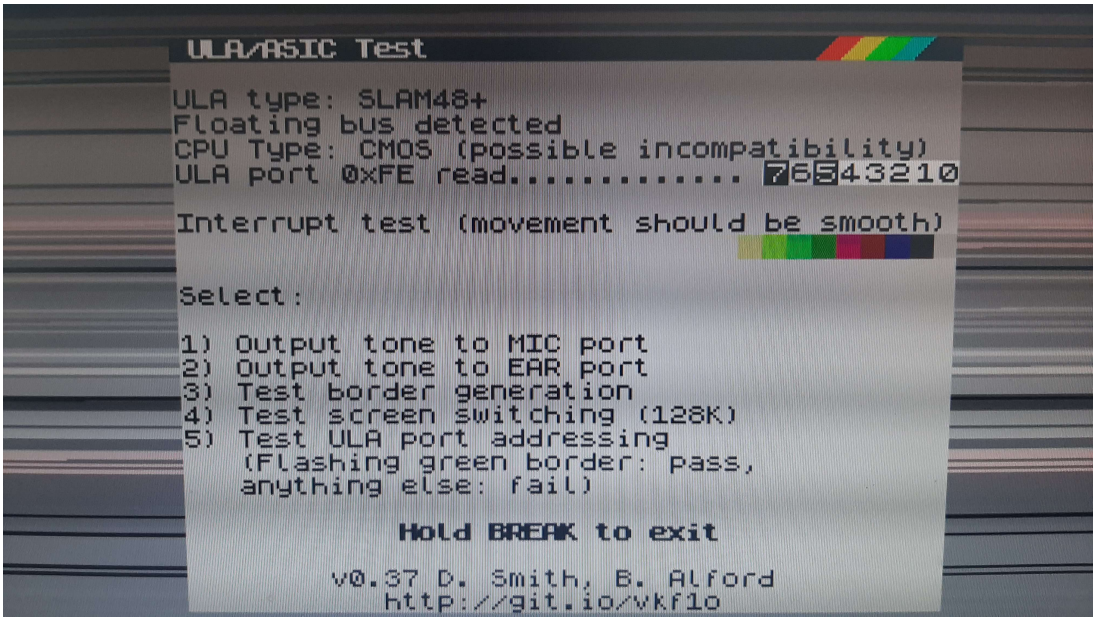

Figure 7.17: ULA test result after implementing the floating bus effect and CMOS cpu type detected

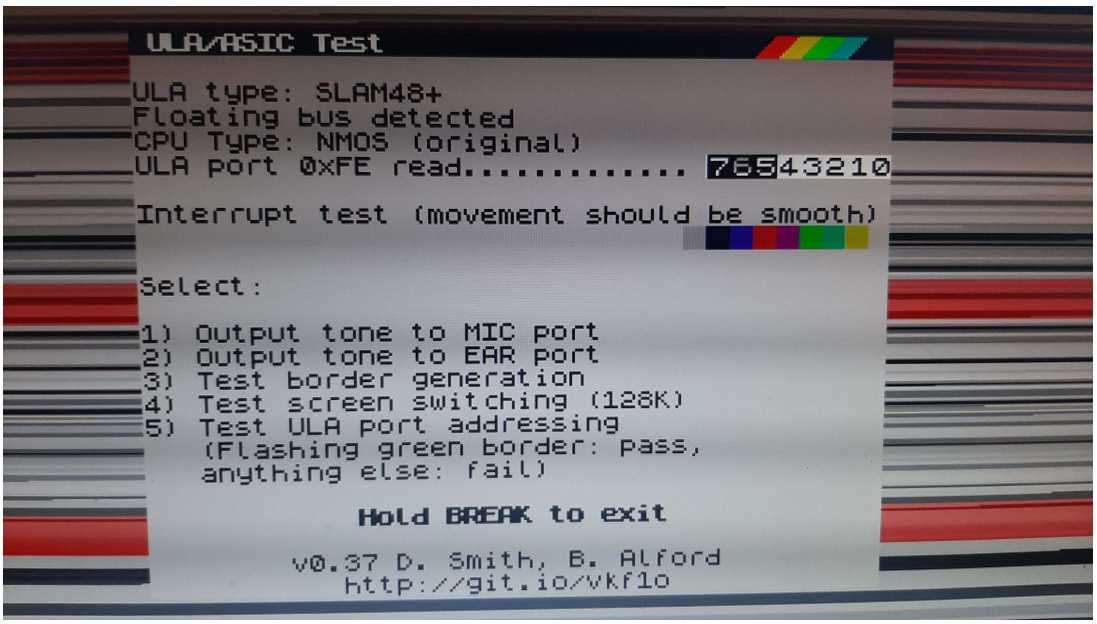

Figure 7.18: ULA test result after implementing the floating bus effect and NMOS CPU type detected

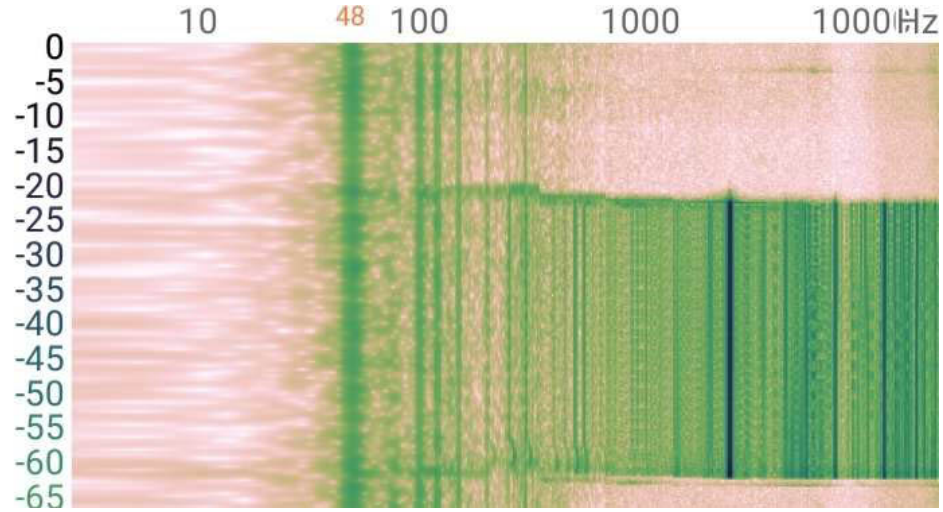

Figure 7.19: Spectogram of the EAR port tone from the ULA test page

8

Conclusions and Future Work

This work's initial objectives were to implement a modernized ZX Spectrum with modern peripherals. Throughout the project, VGA video output, PS/2 keyboard input, audio I/O, SD card support, and joysticks were successfully implemented and validated. Thus, it is possible to state that the main objectives were achieved.

The implementation can be improved on by implementing the unimplemented optional feature mentioned in this report: support for online game repository. Internet access would be a great feature for streamlining the user's experience by obtaining the software directly from broader sources of programs and games.

Another future work could be supporting other models of the ZX Spectrum, like the 128k model. Adding support for such models will facilitate support for a wider range of software that would not work in 48k or 16k models. A possibility is to have the 128k's menu adapted to include the ZX Fusion's menu as a submenu, where the SD loader and online features could be found. The hardware design would need to be updated to instantiate more memory modules and the paging mechanism.

Early computer systems are simple enough to help teach the basics of computer architectures, but capable of maintaining the interest of many by running complex programs such as arcade games. Since the DE2-115 board may be expensive for most students and hobbyists it may be worth porting this work to an FPGA development board with similar peripherals but with less cost. Such a board should at least support VGA/HDMI output, PS/2 or USB keyboard, audio I/O, SD Card, and I/O connector

to connect the joysticks. One such alternative worth investigating is the Altera DE10-lite from Terasic. The FPGA device on this board has enough resources to implement the system. However, it lacks the presence of some peripherals such as the PS/2 keyboard connector, the SD card slot, and the audio I/O. This can be made on a custom Arduino shield, supported by the DE10-Lite board.

As it stands, this project serves as an example of how the NIOS processor can be used to update old systems with features that they would not be able to support otherwise.

References

[1] Paul Farrow. "Spectrum 128 rom". (), [Online]. Available: http://www.fruitcake.plus.com/Sinclair/Spectrum128/ROMDisassembly/Spectrum128ROMDisassembly.htm (visited on 07/28/2023).

[2] ISO/IEC 25010, *ISO/IEC 25010:2011, systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models*, 2011.

[3] "Peripherals". (Jul. 2004), [Online]. Available: https://worldofspectrum.org/faq/reference/peripherals.htm (visited on 08/05/2023).

[4] "16k / 48k zx spectrum reference". (Sep. 2005), [Online]. Available: https://worldofspectrum.org/faq/reference/48kreference.htm (visited on 08/04/2023).

[5] "The lil old zx spectrum 48k service manual". (), [Online]. Available: https://www.1000bit.it/support/manuali/sinclair/zxspectrum/sm/supp2.html (visited on 08/04/2023).

[6] Goran Devic. "A-z80 cpu". (Dec. 2014), [Online]. Available: https://opencores.org/projects/a-z80 (visited on 09/06/2023).

[7] "What is an fpga?" (), [Online]. Available: https://www.arm.com/glossary/fpga (visited on 08/05/2023).

[8] Richard Dymond. "The 'beeper' subroutine". (2022), [Online]. Available: https://skoolkid.github.io/rom/asm/03B5.html (visited on 08/25/2023).

[9] Nico Kaiser. "Sinclair zx spectrum 48k". (May 2012), [Online]. Available: https://commons.wikimedia.org/wiki/File:Sinclair_ZX_Spectrum_48k_%287160141482%29.jpg (visited on 09/21/2023).

[10] Bill Bertram. "Sinclair zx spectrum+ 8-bit computer (1984)". (Aug. 2006), [Online]. Available: https://commons.wikimedia.org/wiki/File:ZX_Spectrum%2B.jpg (visited on 08/04/2023).

[11] Altera Corporation, "Cyclone iv device handbook, volume 1", Nov. 2009.

[12] Andy Dansby. "Using the keyboard". (Jul. 2018), [Online]. Available: https://zxspectrumcoding.wordpress.com/2018/07/07/using-the-keyboard/ (visited on 08/04/2023).

[13] Altera. "Altera de2-115 development and education board - resources". (), [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=139&No=502&PartNo=4#contents (visited on 08/05/2023).

[14] Ferranti, *Ferranti Quick Reference Guide, ULA 1984*. British Telecom Journal, 1983. [Online]. Available: https://archive.org/details/FerrantiQ.RefULA1984/page/n2/mode/1up.

[15] "Floating bus". (May 2021), [Online]. Available: https://sinclair.wiki.zxnet.co.uk/wiki/Floating_bus (visited on 09/04/2023).

[16] Mohd Nazrin Md Isa and Sohiful Anuar Zainol Murad, "Field programmable gate array (fpga): From conventional to modern architectures", in Jan. 2015, page 53, ISBN: 978-967-5415-98-2.

[17] Will Green. "Video timings: Vga, svga, 720p, 1080p". (Jun. 2020), [Online]. Available: https://projectf.io/posts/video-timings-vga-720p-1080p/ (visited on 08/05/2023).

[18] Hardware Bee. "Introduction to gate array". (), [Online]. Available: https://hardwarebee.com/introduction-to-gate-array/ (visited on 08/03/2023).

[19] Don Thomasson, *The Ins and Outs of the Timex TS1000 & ZX81*. England: Melbourne House, 1983, page 14.

[20] Jarek Adamski. "Kempston joystick". (Dec. 2009), [Online]. Available: https://8bit.yarek.pl/hardware/zx.joystick/index.html (visited on 08/05/2023).

[21] Zilog, "Z80 family, cpu user's manual", 2004.

[22] Michael Slater. "Oral history panel on the development and promotion of the zilog z8000 microprocessor". (), [Online]. Available: https://archive.computerhistory.org/resources/access/text/2015/06/102658075-05-01-acc.pdf (visited on 08/04/2023).

[23] Altium. "Ps2 keyboard scan codes". (Jul. 2023), [Online]. Available: https://techdocs.altium.com/display/FPGA/PS2+Keyboard+Scan+Codes (visited on 09/30/2023).

[24] "Reverse-u16". (Apr. 2019), [Online]. Available: https://github.com/mvvproject/ReVerSE-U16 (visited on 09/06/2023).

[25] "What is the difference between sd, sdhc and sdxc cards?" (), [Online]. Available: https://www.transcend-info.com/Support/FAQ-930 (visited on 08/05/2023).

[26] Cactus Tech. "An introduction to sd card interface". (), [Online]. Available: https://www.cactus-tech.com/wp-content/uploads/2019/03/An-Introduction-To-SD-Card-Interface.pdf (visited on 08/05/2023).

[27] C. Smith, *The ZX Spectrum ULA: How to Design a Microcomputer* (ZX Design Retro Computer). ZX Design and Media, 2010, ISBN: 9780956507105. [Online]. Available: https://books.google.pt/books?id=IMPTcQAACAAJ.

[28] "Sna format". (May 2021), [Online]. Available: https://sinclair.wiki.zxnet.co.uk/wiki/SNA_format (visited on 07/27/2023).

[29] Matt Westcott. "The speccy2010: A complete guide for non-russian-speakers". (Feb. 2011), [Online]. Available: https://web.archive.org/web/20170706104143/http://matt.west.co.tt/spectrum/speccy2010/ (visited on 09/06/2023).

[30] Istvan Novak. "Spectnet ide, fast (instant) load". (Jul. 2023), [Online]. Available: https://dotneteer.github.io/spectnetide/getting-started/fast-load (visited on 09/30/2023).

[31] "About - zx spectrum next". (2017), [Online]. Available: https://www.specnext.com/about/ (visited on 09/06/2023).

[32] "File formats - specnext official wiki". (Nov. 2019), [Online]. Available: https://wiki.specnext.dev/File_Formats (visited on 09/06/2023).

[33] Stephen Eddy. "Spectrum emulator for terasic de-115". (Jul. 2023), [Online]. Available: https://unashamedgeek.com/post/105367852025/spectrum-emulator-for-terasic-de-115 (visited on 09/30/2023).

[34] Mike Stirling. "Sinclair zx spectrum 48k and 128k on an altera de1 fpga board". (Jan. 2016), [Online]. Available: https://github.com/mikestir/fpga-spectrum (visited on 09/30/2023).

[35] Richard Dymond. "The complete spectrum rom disassembly". (Jul. 2023), [Online]. Available: https://skoolkid.github.io/rom/index.html (visited on 07/18/2023).

[36] Daniel; MikeJ Wallner. "T80 cpu". (Apr. 2002), [Online]. Available: https://opencores.org/projects/t80 (visited on 09/06/2023).

[37] Miguel Angel Rodriguez Jodar. "Ula chip for zx spectrum". (Apr. 2012), [Online]. Available: https://opencores.org/projects/zx_ula (visited on 09/06/2023).

[38] EECG University of Toronto. "Ps/2 controller". (Jul. 2023), [Online]. Available: https://www.eecg.utoronto.ca/~jayar/ece241_08F/AudioVideoCores/ps2/ps2.html (visited on 07/02/2023).

[39] Scott Larson. "Vga controller (vhdl)". (Mar. 2021), [Online]. Available: https://forum.digikey.com/t/vga-controller-vhdl/12794 (visited on 09/27/2023).

[40] SECONDS Ltd. "Xga signal 1024 x 768 @ 60 hz timing". (2008), [Online]. Available: http://tinyvga.com/vga-timing/1024x768@60Hz (visited on 09/27/2023).

[41] *Portable internet audio codec with headphone driver and programmable sample rates*, WM8731, Rev. 4.9, Wolfson Microelectronics, Oct. 2012.

[42] "Your sinclair 20". (Aug. 1987), [Online]. Available: https://worldofspectrum.org/archive/magazines/your-sinclair/20/#56 (visited on 08/04/2023).

[43] "Z80 file format". (Jul. 2004), [Online]. Available: https://worldofspectrum.org/faq/reference/z80format.htm (visited on 07/27/2023).

[44] Rob Landley. "The z80 microprocessor". (Aug. 1998), [Online]. Available: https://landley.net/history/mirror/cpm/z80.html (visited on 08/04/2023).

[45] Samuel McConnel. "Z80: The most prolific cpu of all time". (Mar. 2016), [Online]. Available: https://www.kmuw.org/your-move/2016-03-10/z80-the-most-prolific-cpu-of-all-time (visited on 08/04/2023).

[46] Andy Karpov. "Zx-ula-wxeda: Zx spectrum 48k with ulaplus". (Nov. 2022), [Online]. Available: https://github.com/andykarpov/zx-ula-wxeda (visited on 11/09/2023).

[47] Albert Veli. "Zx_beep". (Jul. 2013), [Online]. Available: https://github.com/AlbertVeli/ZX_Beep (visited on 08/04/2023).

[48] Rui Ribeiro. "O beeper do spectrum". (Nov. 2019), [Online]. Available: https://planetasinclair.blogspot.com/2019/11/o-beeper-do-spectrum.html (visited on 08/04/2023).

[49] brendanalford; ZXGuesser. "Brendanalford/zx-diagnostics". (), [Online]. Available: https://github.com/brendanalford/zx-diagnostics (visited on 09/01/2023).

[50] brendanalford; ZXGuesser. "Firmware - zx spectrum diagnostis". (), [Online]. Available: https://github.com/brendanalford/zx-diagnostics/wiki/Firmware (visited on 09/07/2023).

[51] "Zx spectrum hardware description". (Mar. 2014), [Online]. Available: https://youtu.be/9-lA2F2uyA4?t=1068 (visited on 08/04/2023).

[52] "The sinclair zx interface 1". (), [Online]. Available: http://www.retro8bitcomputers.co.uk/Sinclair/ZXInterface1 (visited on 09/21/2023).

[53] Chris Owen. "Interface 2". (), [Online]. Available: https://rk.nvg.ntnu.no/sinclair/computers/peripherals/interface2.htm (visited on 08/05/2023).

[54] Dean Beltfield. "Memory map". (), [Online]. Available: http://www.breakintoprogram.co.uk/hardware/computers/zx-spectrum/memory-map (visited on 08/04/2023).

[55] "Memory contention and the floating bus". (), [Online]. Available: https://spectrumforeveryone.com/technical/memory-contention-floating-bus/ (visited on 09/04/2023).

[56] Dean Beltfield. "Motherboard". (), [Online]. Available: http://www.breakintoprogram.co.uk/hardware/computers/zx-spectrum/hardware (visited on 08/04/2023).

[57] Jonathan Harston. "Zx spectrum rom images". (Sep. 2004), [Online]. Available: https://mdfs.net/Software/Spectrum/ROMImages/ (visited on 08/05/2023).

[58] Dean Beltfield. "Screen memory layout". (), [Online]. Available: http://www.breakintoprogram.co.uk/hardware/computers/zx-spectrum/screen-memory-layout (visited on 08/04/2023).

[59] Dean Beltfield. "Sound". (), [Online]. Available: http://www.breakintoprogram.co.uk/hardware/computers/zx-spectrum/sound (visited on 08/04/2023).

[60] "Zx-uno [zx spectrum computer clone based on fpga]". (Oct. 2013), [Online]. Available: https://zxuno.speccy.org/maquina_e.shtml (visited on 09/06/2023).

[61] Dave Curran. "Sinclair zx80 repair". (Feb. 2013), [Online]. Available: http://blog.tynemouthsoftware.co.uk/2013/02/sinclair-zx80-repair.html (visited on 08/04/2023).

[62] Dave Stevenson. "Zx80 & zx81 comparison". (), [Online]. Available: http://www.primrosebank.net/computers/zx80/zx80_vs_zx81.htm (visited on 08/04/2023).

User Manual

by Gustavo Jacinto

September 29, 2023

Contents

1 Introduction

The ZX Spectrum Fusion was implemented on a DE2-115 FPGA board. This manual describes how to set up the implementation for this board to run original ZX Spectrum games.

2 Setting Up the ZX Spectrum Fusion

This section describes how you can set up the implementation on an DE2-115 development board from Terasic. To set up the Spectrum Fusion, the following accessories are required:

- DE2-115 board and its power cable;
- USB Type B to Type A cable;
- A computer with Intel Quartus Prime installed;
- VGA cable and monitor;
- PS/2 keyboard;
- A 3.5mm audio cable M/M and a speaker;
- A 3.5mm audio cable M/M and an audio cassette player or a phone;
- (Optional) An SD card with .z80 or .sna files in the root directory;
- (Optional) The expansion board to connect an original ZX Spectrum keyboard and up to two controllers;
- (Optional) 1 or 2 serial joysticks (NES clones).

2.1 How to Connect the Peripherals

Figure 1 shows a top view of the DE2-115 board. The peripherals can be connected according to the figure:

- A. VGA cable to a VGA monitor;
- B. PS/2 keyboard;
- C. 3.5mm male-to-male audio cable to an audio speaker;
- D. audio input for a device that can play audio, such as a tape player or a smartphone, using a 3.5mm male-to-male audio cable;
- E. SD card slot. The slot is located under the board in that location;
- F. Pin header for connecting the expansion board;

Pictured in Figure 2 is the expansion board, where you can connect, from left to right, joysticks 2 and 1 and a native ZX Spectrum keyboard.

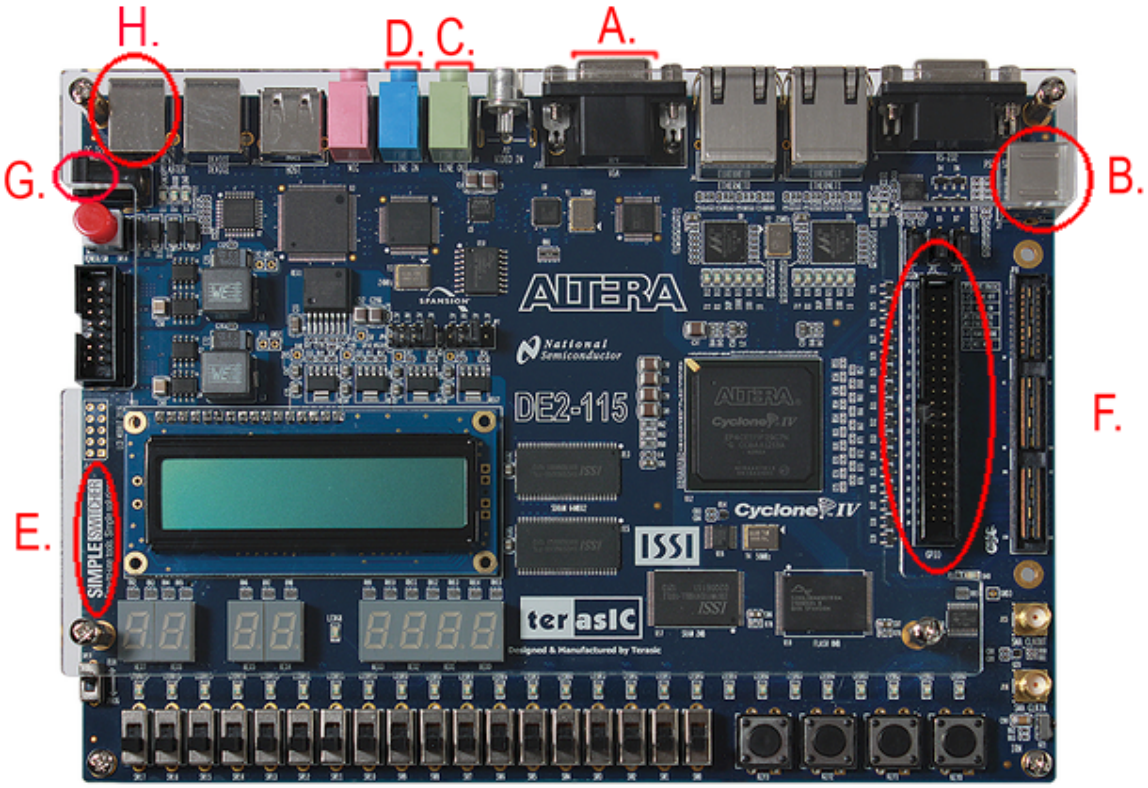

Figure 1: A DE2-115 board with the relevant ports highlighted.

Figure 2: A DE2-115 board with the Joystick/Keyboard expansion board connected to it.

Figure 3: ZX Fusion configuration inputs. In this figure, the ZX Fusion is set to operate with a Sinclair joystick (1), a PS/2 keyboard (2), and full-screen video (3). The push-button (4) resets the system.

2.2 Powering the FPGA Board

To power the FPGA board, plug the cable from the power adapter into the slot labeled "G." in Figure 1. Plug the power adapter to a power socket. The red button located next to the slot is the ON/OFF button. Press it once to turn on the board. An image of the board should be displayed on the connected VGA monitor, according to the initial demo that comes with the board.

2.3 Flashing the DE2-115 FPGA Board

The process of flashing the board is present in its user manual (https://www.terasic.com.tw/attachment/archive/502/DE2_115_User_manual.pdf). It is described in Chapter 4, Section 4.1 "Configuring the Cyclone IV E FPGA".

3 Board Input

The board has many switches and buttons on it, and some were configured to have an effect on the ZX Spectrum Fusion. Figure 3 shows the configuration inputs. They can be used to do the following:

1. Toggle between Sinclair and Kempston joysticks. Down = Sinclair and Up = Kempston;
2. Toggle between a PS/2 keyboard and a native ZX Spectrum keyboard. Up = native keyboard and Down = PS/2;
3. Set the video mode. The left switch down sets the screen resolution to 4x, occupying the whole screen except for the 1-pixel wide border. The left switch up sets the resolution to 2x and both of them up sets the resolution to 1x;
4. Reset Push-Button.

4 Loading Programs

This section includes instructions for loading programs, either through the line-in port or the SD Card. On boot, if everything is ok, the menu in Figure **??** should be displayed.

4.1 From Audio Line-in

To load a program from an audio cassette or smartphone (audio source), select the last option in the menu in Figure 4, "BASIC". This will start the original ZX Spectrum

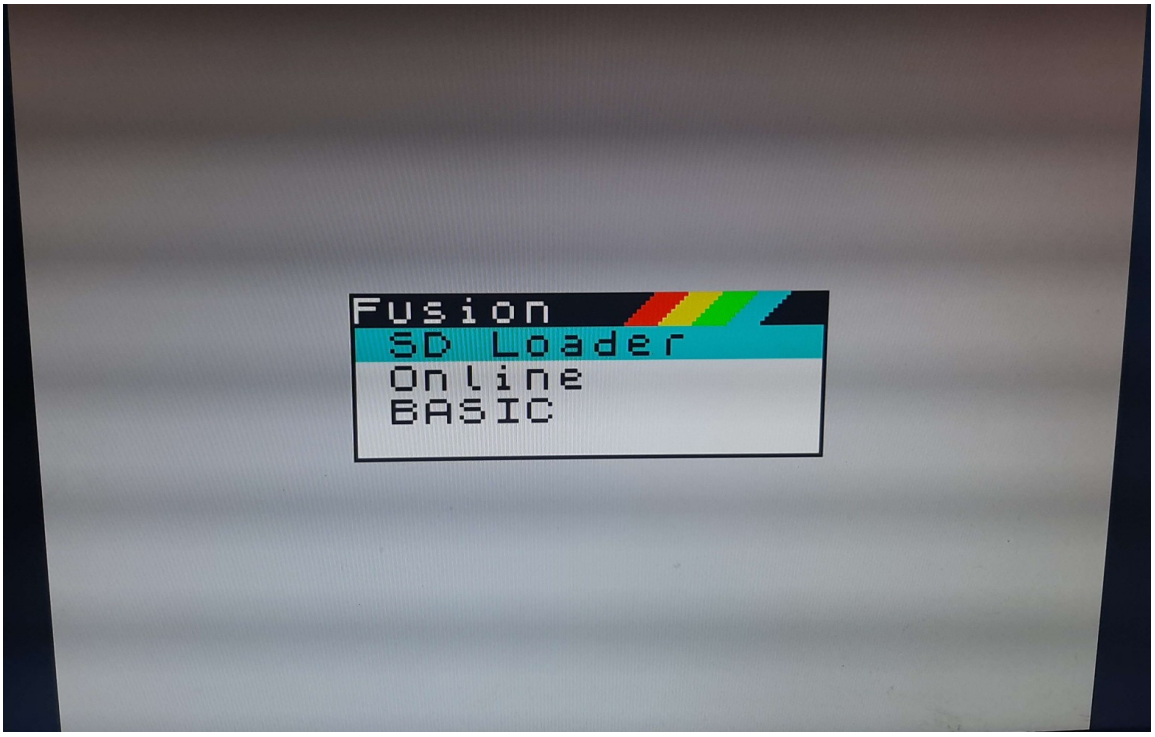

Figure 4: The main menu screen

48k's BASIC editor. Type in `LOAD ""` and press enter. The border should flash red and blue. Now, start the playback of the program audio, and the border should start alternating colors. This means it is loading the program.

4.2 From an SD Card

To load a .z80 or .sna file from an SD Card, select the first option in the menu in Figure 4, "SD Loader". This opens a new menu listing the .z80 and .sna files in your SD Card, as seen in Figure 5. Press ↑ and ↓ keys or joystick buttons to navigate up or down in the menu. Only 16 files appear per page. To change pages, press ← or → on the keyboard or joystick. When the game you want is highlighted, press the ENTER key or the FIRE button on your joystick to select the file. The program will start loading and the rightmost LED on the board will remain ON until it is fully loaded. Bomb Jack II can be seen running in the ZX Fusion in Figure 6.

NOTICE: Joystick for menu navigation only works if you have it in the Sinclair Joystick mode.

5 ZX Fusion Memory Map

The memory map for the ZX Fusion can be seen in Figure 7. The bigger memory regions, ROM, Screen Memory, Color Memory, and Remaining RAM, are separate memory instances on the computer. The lighter and smaller regions contain code used for this implementation's main menu and routine loading.

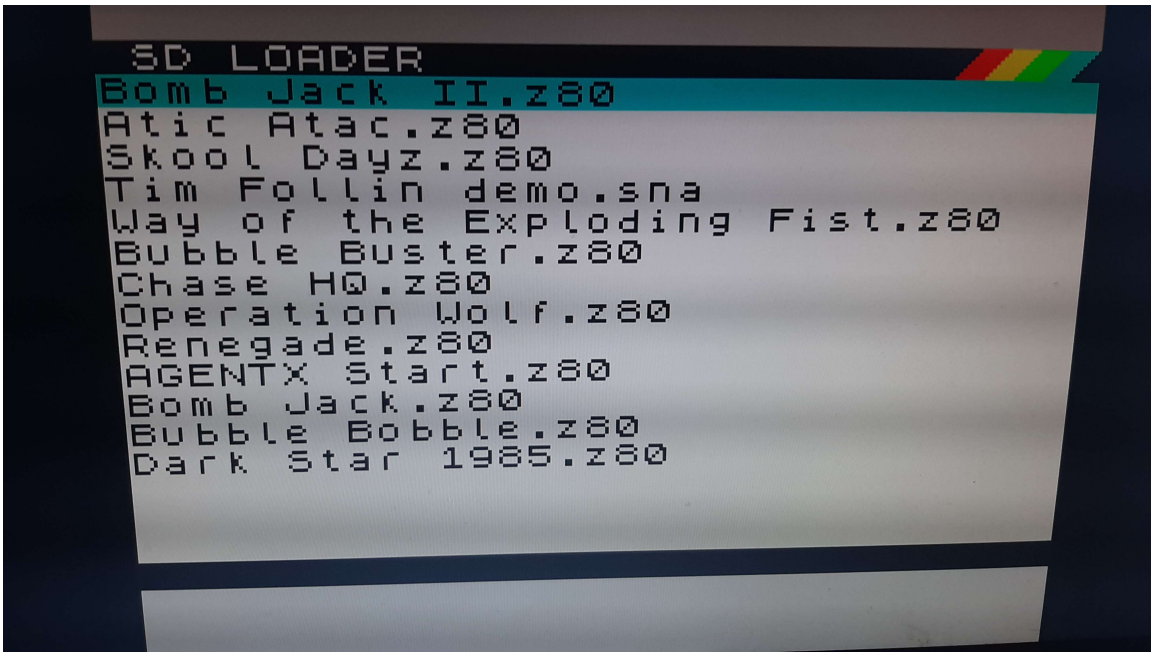

Figure 5: SD Loader listing the files on the SD Card. Bomb Jack II is highlighted

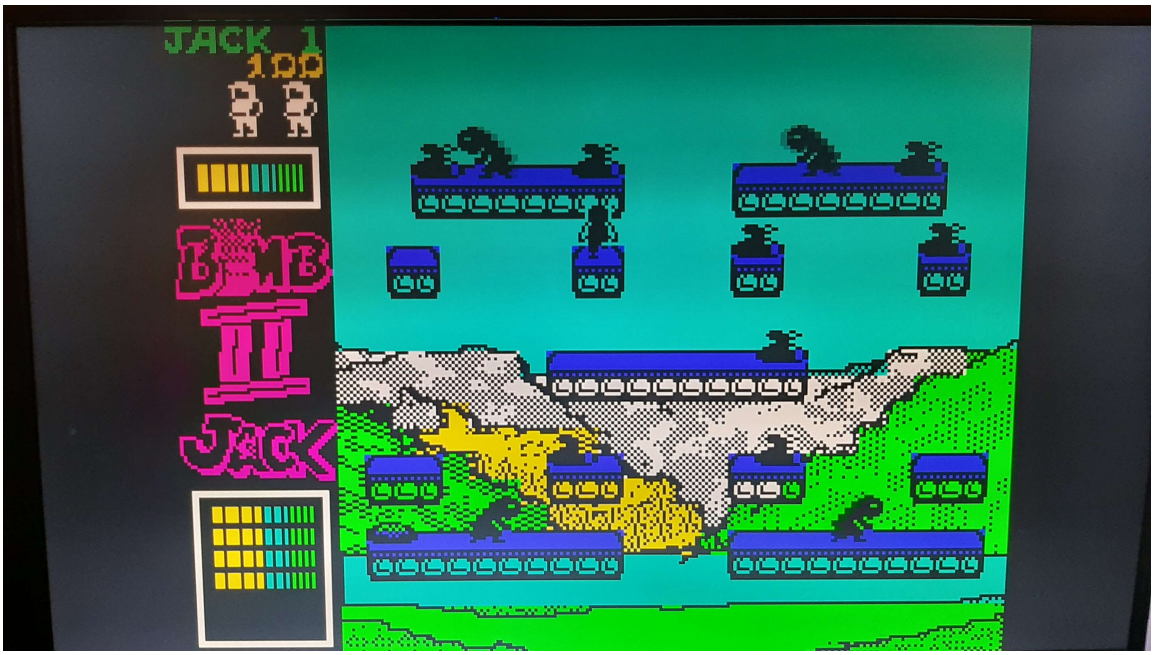

Figure 6: Bomb Jack II in-game

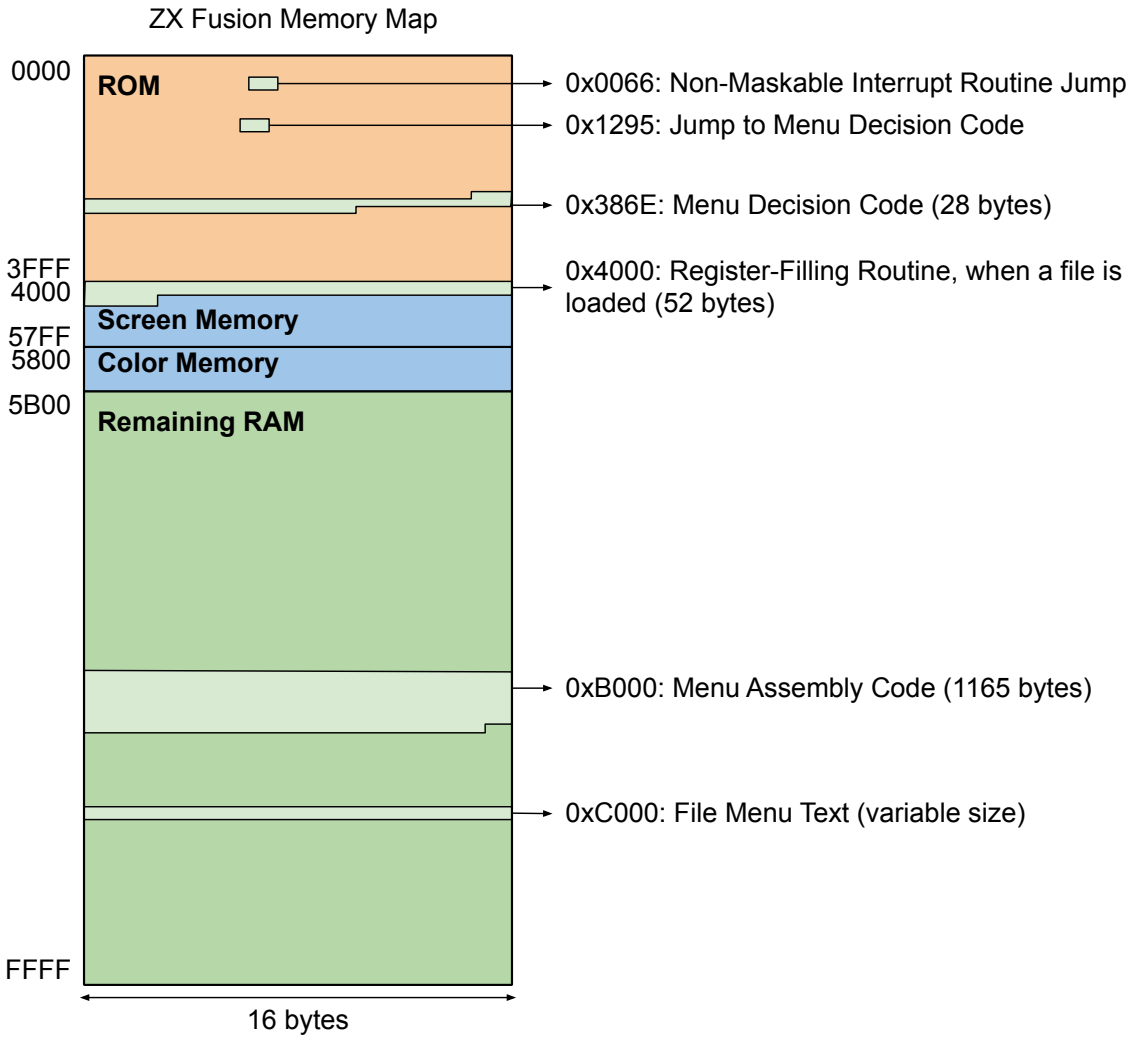

Figure 7: The ZX Fusion's Memory Map

– End of Document –