**Designing a 32-bit FPGA Computer**

By

Youssef Elgedawy

**May 2024**

**1 Abstract**

This project involves designing and implementing a full computer whose heart is a fully programmable 32-bit CPU that is able to control any peripheral by memory mapping control registers. The entirety of this project was done using Intel Quartus Prime and ModelSim, and all FPGA RTL code is fully synthesisable and was written using SystemVerilog. A small summary of the major components of this project are listed below.

**Video Graphics Array (VGA) Controller:** a device that implements the VGA protocol to allow the use of any display to be used for rendering. Despite implementing the VGA protocol, any monitor with HDMI capabilities may be used by using a VGA-to-HDMI adapter.

**PS/2 Controller:** a receiver that understands the PS/2 protocol. This project uses the PS/2 keyboard as a peripheral. However, the use of a PS/2 keyboard and mouse may be used with this controller provided the FPGA development board has 2 PS/2 ports, or alternatively a single port in conjunction with a Y-splitter. The main use of this keyboard was to allow user-input for conditional processing by the CPU and may equally be used to send data.

**Universal Asynchronous Receiver Transmitter (UART):** the “messenger” of the computer – a module that implements the Recommended Standard no.232 protocol (RS232) in order to program the CPU via a single cable. Despite calling it a UART module, the project only implements the device as a receiver and not a transmitter as the computer does not need to send anything back to the transmitter. Programming may be easily done by any MCU that as UART capabilities (most of them!).

**Simple Audio Controller:** described as “simple” as the audio output is a humble PWM buzzer. By sending this controller a specific byte, a particular musical frequency will be output by the buzzer. All these frequencies were stored as an array of frequency dividers from the main 50MHz FPGA clock and a demultiplexer (DEMUX) to choose which frequency to output.

**The Central Processing Unit (CPU):** the “brain” of the computer. The design of this CPU was heavily influenced by the ARMv7 Instruction Set Architecture (ISA) including (as will later be discussed) the assembly mnemonics and how the NZCV flags were implemented. In fact, this CPU has 13 general-purpose registers (r0-r12) for fast access and temporary storage and load and store architecture from a fully custom memory map that includes the program memory, memory mapped registers and even the stack!

**WS28128B Addressable LED controller:** due to the nature of this project, a lot of this system’s beauty is hidden. In an attempt to reveal *some* of that to pleasure the eyes, some crucial internal registers will be connected to this module and to a WS2812B LED strip to allow anyone to see the contents of those flip-flop arrays! A custom PCB was designed and built to accommodate this and supports the 13 general purpose registers (GPRs) and up to 10 memory mapped registers (MMRs).

**2 Acknowledgements**

The author of this dissertation would like to sincerely thank the *former Senior Lecturer (SL) Nicholas Outram* for his guidance during the early planning phase of the CPU design regarding tri-state logic and synthesisable RTL code as without that knowledge, countless hours and efforts would have been spent wondering why the RTL code works during testbenches but is not synthesisable.

*A more detailed discussion about RTL synthesis and FPGA resource utilisation will be mentioned later during the dissertation.*

**Contents**

**3 Abbreviations**

**4 Introduction**

Embedded systems and electronics tend to always be seen as this “magical black IC box” that does what it does, without knowing how it does it. From the simplest ICs like operational amplifiers to infinitely more complex systems like CPUs and MCUs, they are all seen that way. The motivation behind this project was to open that “magical box” up and reveal not only its contents but – quite frankly – its beauty; to truly understand the inner workings of those silent and stationary mechanisms and thus appreciating the efforts of those that came before us to bring us what we rely on today.

The computer: a system that most people on this planet have used or at least come across. Doesn’t such a fundamental member of society deserve to be understood? Understanding the workings of a computer allows one to study the current architectures and think of ways to improve them – make them more efficient; more powerful. Which is such a crucial question to ask as these computers are one of the many backbones of society and technological advancements today.

In order to design and build complicated systems, simpler versions must be constructed or at least fully understood, which is precisely the approach the author took on this project. A barebones CPU, with merely 4 instructions and a few basic registers was first planned, tested and synthesized on an FPGA. Then, using the skills gathered from that mini-project, the final CPU was built, with 16 of the most used instructions today from the Reduced Instruction Set Computer (RISC) architecture.

**5 Project Management**

The project inaugurated on the beginning of February 2024, and was completed mid-April 2024. The two and a half months the project spanned were extremely intensive as lots of work had to be done to get the many components working together harmoniously. Hence management was crucial. Initially, this project was worked on 4 days a week (Tuesday-Friday) for anywhere between 4-6 hours each day. Later on (after a week break from the project before the Easter break) the project was worked on daily for 5-8 hours per day.

The plan was to construct all the peripherals first and get them in a fully working condition in order to gauge the CPU ISA required to control these peripherals. *The approach taken here was to build the CPU around the specific requirements of the system.* In order to realise this, one working week was spent implementing each peripheral (UART, VGA controller, PS/2 controller and the audio controller). This takes us to about the end of February. So a month and a half was spent researching and implementing two CPU architectures (a simple one for learning, and the final custom CPU).

During the final phase of the project, the idea of constructing some kind of LED panel popped up and many different ideas/revisions were tried until the final iteration (WS2812B LED array) was achieved. This in total took around 1 week.

**6 Research**

Naturally, as very minimal background knowledge was had prior to starting the project, research had to be done on each component and protocol before beginning.

**6.1 The VGA Protocol**

Fundamentally speaking, this protocol consists of two counters that control two synchronisation signals the Vertical Synchronisation (VSYNC) signal and the Horizontal Synchronisation (HSYNC) signal. The protocol will be explained with the aid of the diagram below

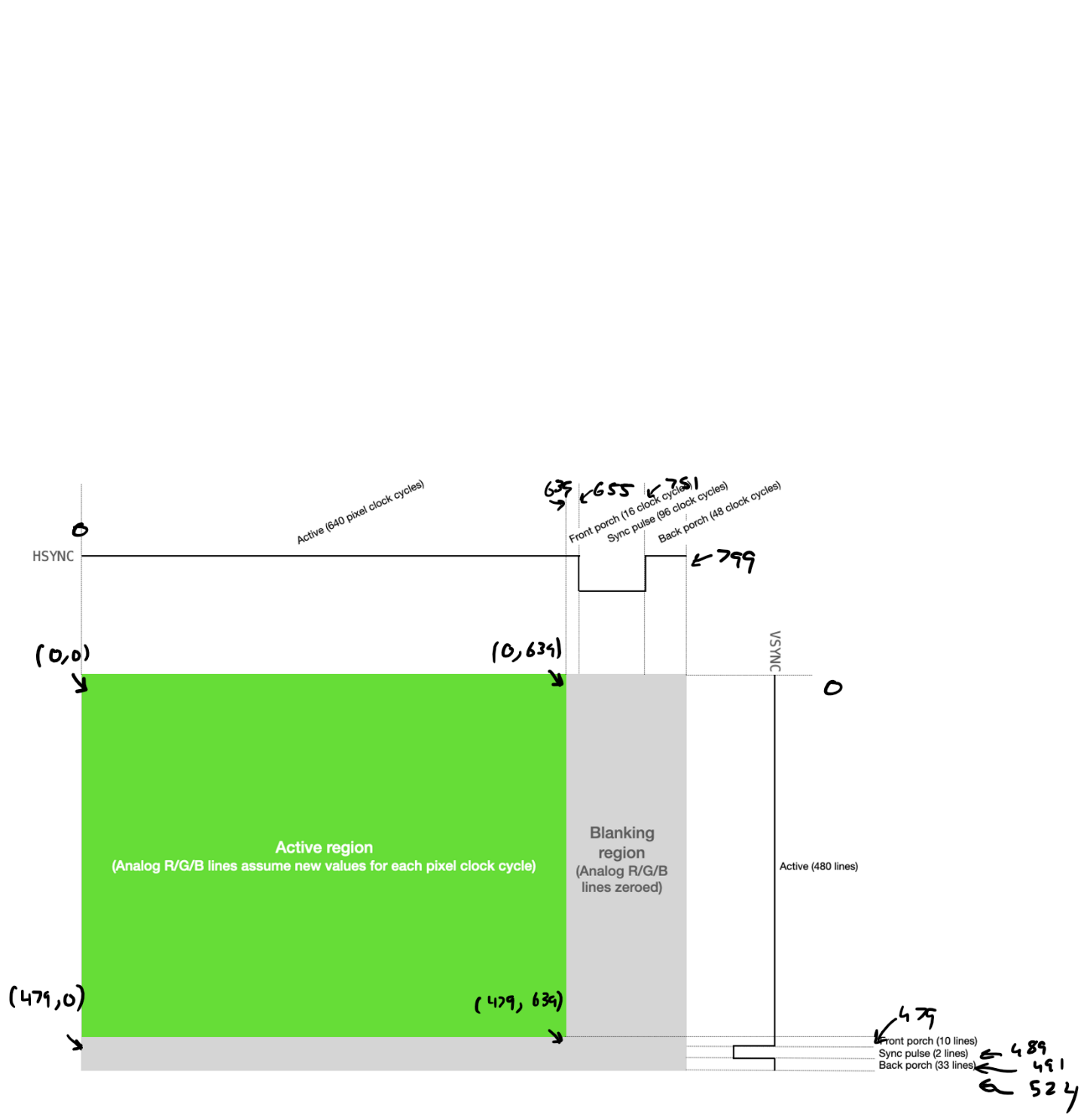


Figure 1 – VGA protocol diagram for a 640x480 resolution [1] with the authors own annotations where the numbers in brackets represent *(rows,columns)*

Before talking about the protocol, an important note must be made. The clock frequency that drives the VGA controller module (which controls the HSYNC, VSYNC and complementary components) must run at very specific clock frequencies (known as the pixel clock) according to the desired resolution and refresh rate. See the table on the next page.

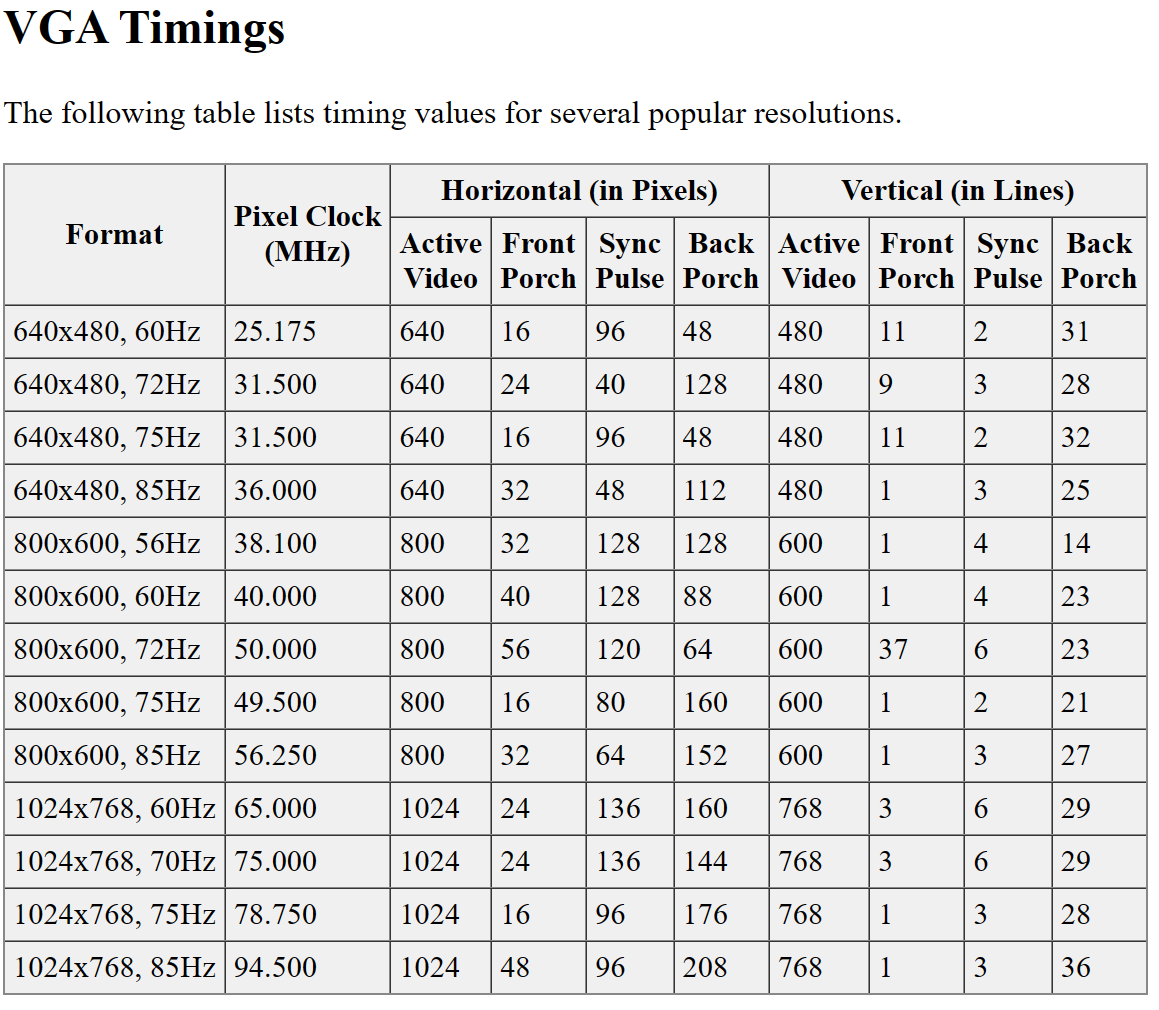


Figure 2 – table of pixel clock frequencies according to resolution and refresh rate [2] *it may be noted that slight deviations of those frequencies will still satisfy the protocol implementation*

The key parts of the protocol are as follows:

* Active region – the region in the display that the viewer sees the image in (in this case it is 640 columns by 480 rows)
* Blanking region – this is when no colour is displayed and the scan line approaches the end of the row/frame. This region is divided into three parts: 1) front porch, 2) back porch and 3) synchronisation pulse
* HSYNC and VSYNC synchronisation pulses: synchronises the start of the horizontal picture scan line in the monitor with the picture source that created it. VSYNC is the equivalent vertical synchronisation, it ensures the monitor scan starts at the top of the picture at the right time [3].

**Colour**

So far, if the timings are implemented correctly, the display will switch on and detect a graphics controller communicating in the VGA protocol and will automatically set the resolution and frame rate, however it will just be a blank screen



Figure 3 – lit monitor with no RGB signal [4]

The VGA port is known as the DB-15 connector. Most of these pins are ground or not connected (NC). The pinout are shown below

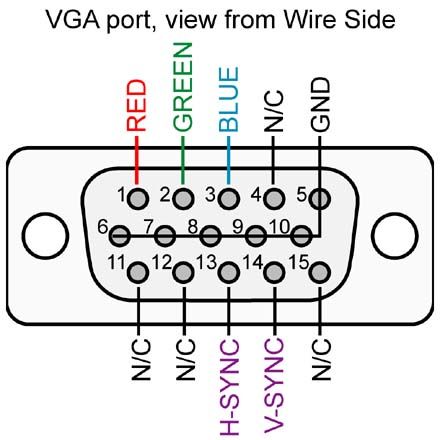


Figure 4 – DB-15 pinout [5]

In order to display colour on the screen, variations of the RGB voltages on the pins will display different colours. But a problem arises, the VGA protocol is an analogue one where the intensity of colours is controlled by controlling the voltage on the RGB pins from 0V up to 0.7Vpp [6], and the FPGA is in the digital domain. If the FPGA where to be connected to the RGB pins directly, there will be no variation in the intensity of the RGB colours and they will either be on or off. To fix this issue, an N-bit resistor ladder digital-analogue-converter (DAC) network may be used. The Terasic DE0-CV development board [7] that was used in this project had a 12-bit DAC resistor network embedded in it as shown below

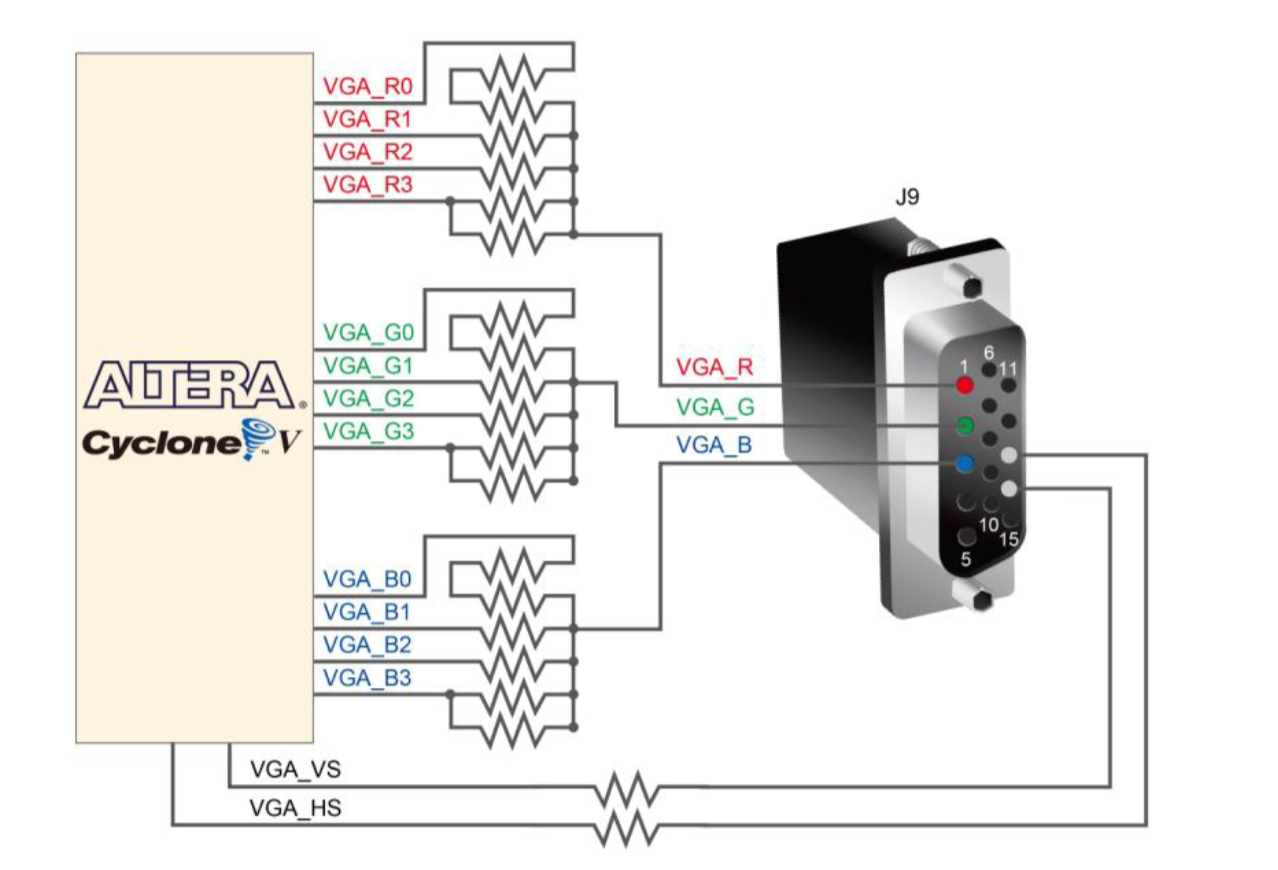


Figure 5 – DB-15 pinout on the DE0-CV board, taken from its user manual [8]

So now, 12-bit colour is available for the user to use; meaning that there are 4-bits to control the intensity of each colour where a value of 0x00 is off and 0x0F is maximum intensity of one colour. Thus, in RGB order: 0x000 (all colours off) and 0xFFF (all colours on – white!).

**6.2 The RS232 protocol**

The UART module is a device that implements this protocol: a serial asynchronous protocol that uses the concept of data framing and other techniques to ensure data synchronisation. The bare minimum to get a transmitter (Tx) and receiver (Rx) to start communication *with a single wire* via this protocol is:

* **Agreement on baud rate (data rate in bits/second – bps):** both the Tx and the Rx must know what baud rate to communicate with
* **Knowledge of data frame structure:** both devices must know the voltage level of the start bit, the size of the data (8-bits or 9-bits), whether or not a parity bit will be used and its type (odd/even) and the voltage level of the stop bit

Other techniques to ensure reliable communication such using a USART (synchronous UART) or the use of acknowledgement was not implemented in this project and will not be discussed, but is good for the reader to be aware of.

The output of the UART Tx differs slightly depending on whether or not a line driver is used. Consider the figure below.

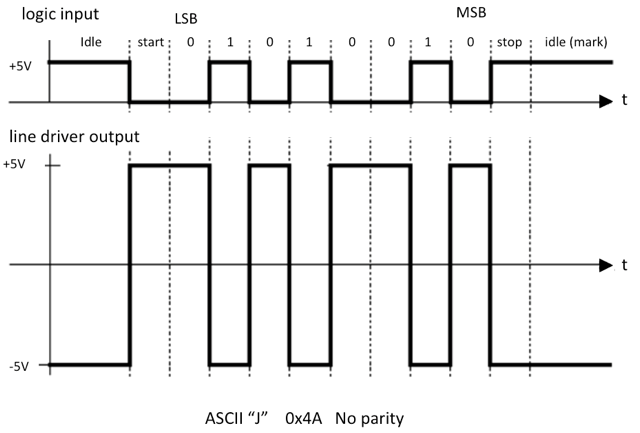


Figure 6 – RS232 frame structure. Top: logic input (and output if no line driver is used). Bottom: line driver output [9]

In both cases one thing to note is that the least significant byte (LSB) is sent first. However, the main difference is the voltage levels and the logical inversion (where a 1 is a low voltage level and vice versa). The UART Tx to be used is the NUCLEO STM32F429ZI MCU development board [10]. It was found using an oscilloscope that no line driver is used on its output of the UART pins. If so, intermediary analogue circuitry would be required to shift the voltage levels down to the Transistor-Transistor Level (TTL) voltage standards (an example design will be discussed later).

**6.3 The PS/2 protocol (Keyboard)**

This protocol is also a serial one that, upon inspecting the waveforms is very similar to the RS232 protocol but with a few key differences.

1. A clock is required along with the serial data. The PS/2 clock is generated by the host and is idle (high) by default. This clock is generated once a byte of information needs to be sent. Only then, is the clock line pulled low then a clock is generated until the end of the data frame.
2. The data frame structure also consists of a low start bit and a high start bit. But the data itself is only 8-bits and an odd parity bit is also included.
3. The byte sent is **not the ASCII representation** of the character sent, instead this byte is known as a *scan code.*

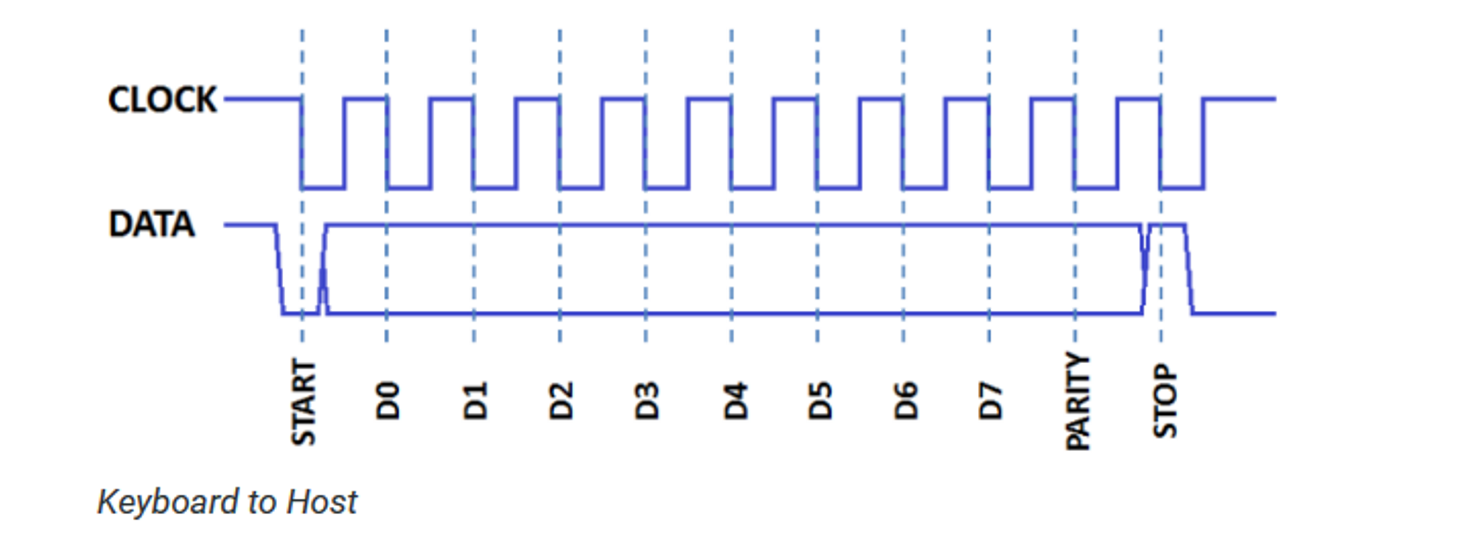


Figure 7 – PS/2 keyboard frame structure [10]

**The Scan Code**

When a key is pressed a *make code* is generated, and when a key is released a *break code* is generated. Most keyboard use what is known as a *set 2 scan code*, meaning that the break code is the same as the make code followed by 0xF0. In this type of scan code, most of the keys pressed will generate a 1-byte make code, however some keys use an *extended make code* in which the byte 0xE0 is sent first, followed by its make code. See the scan code list on the next page.

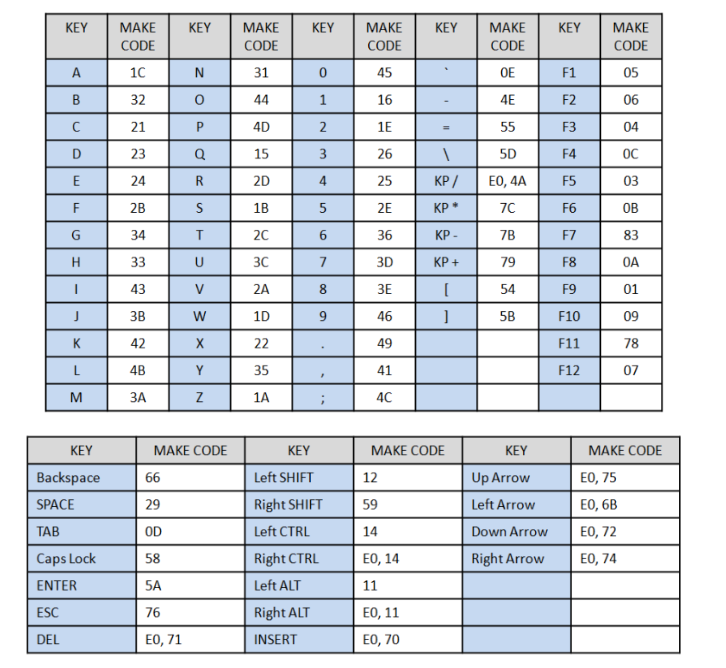


Figure 8 – PS/2 keyboard scan codes [11]

In this project, the keyboard is the “host”, and the FPGA is the “device”. So far only the data sent from host to device has been covered. The protocol of sending special bytes (commands) from device to host was not implemented and will not be explained but a table of commands are listed below

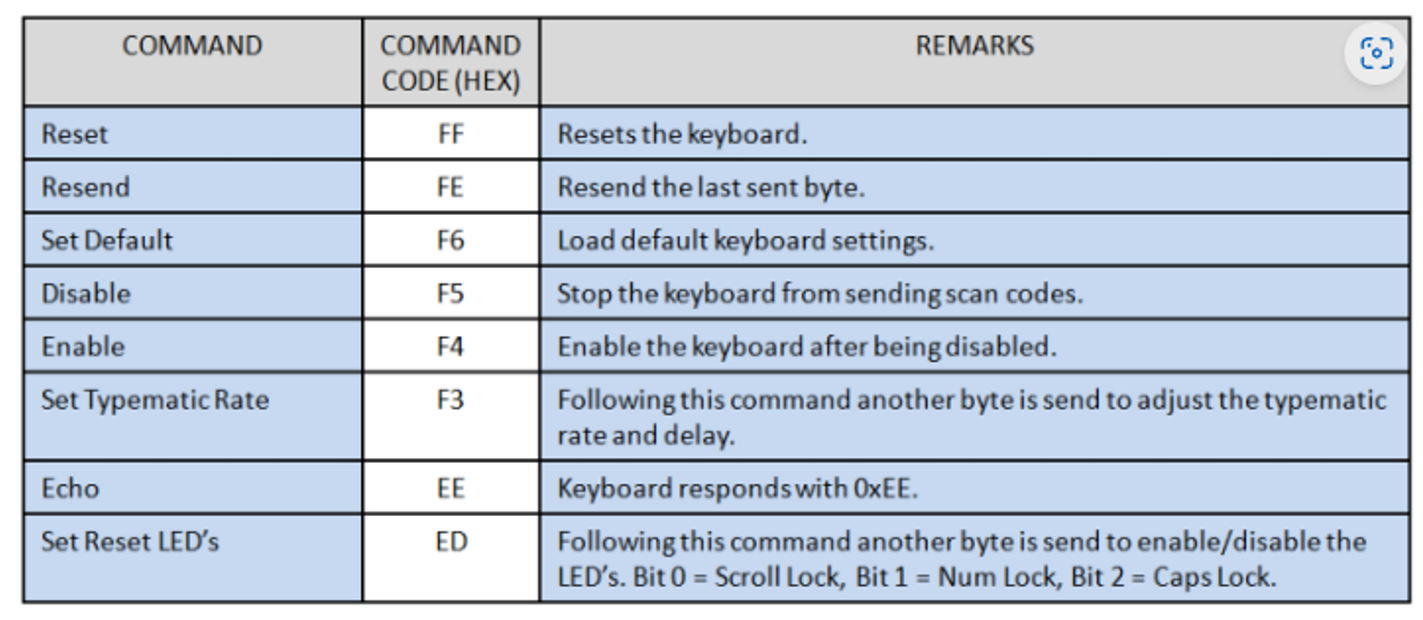


Figure 9 – PS/2 keyboard hex commands sent from “device” to “host” [11]

**6.4 The Audio Controller**

The audio output device for the computer was to be a simple PWM buzzer, which oscillates at a frequency equal to the input voltage frequency applied to its terminals. As musical notes were planned to be played later on in the project, the specific frequencies of these notes must be known.

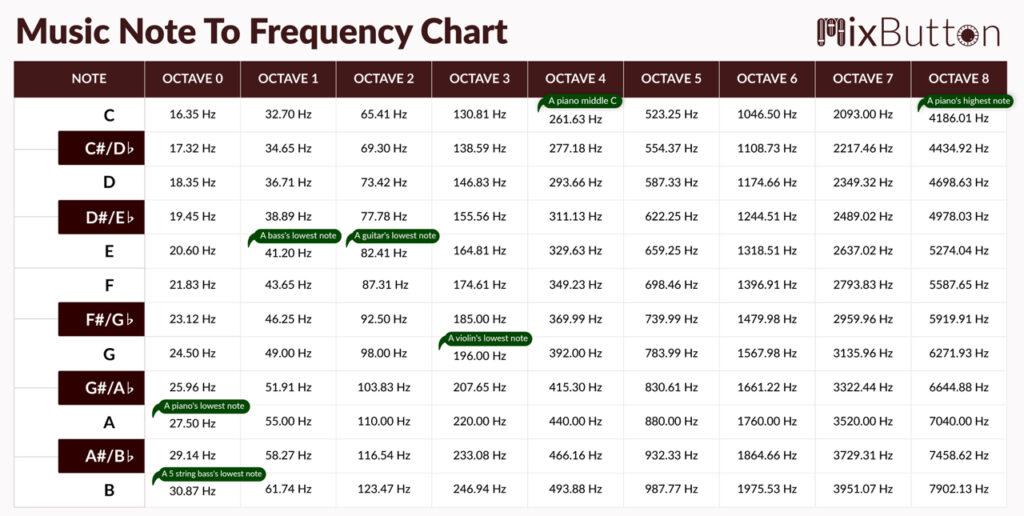


Figure 9 – musical notes frequency mapping [12]

**6.5 The CPU**

*The research of CPU architecture in this section was done entirely by following a textbook whose author could not be found, but the extract was from Chapter 6: CPU Design [11].*

In general, a CPU performs the following sequence of operations:

* **Fetch cycle:** fetch an instruction from memory, then go to the decode cycle.
* **Decode cycle:** decode the instruction—that is, determine which instruction has been fetched—then go to the execute cycle for that instruction.
* **Execute cycle:** execute the instruction, then go to the fetch cycle and fetch the next instruction.

The CPU has a few fundamental registers that it uses for operation:

* **Address Register (AR):** stores the address of the current instruction to be fetched from memory.
* **Program Counter (PC):** stores the address of the next instruction to be fetched from memory.
* **Data Register (DR):** stores the entire instruction “word” from memory, which consists of the opcode, operands, and any other bits to be used for a specific reason.
* **Accumulator (AC):** stores the result of any arithmetic and logic operation on operands (output from the ALU)
* **Instruction Register (IR):** stores the opcode portion of the instruction word

**A diagram of a computer

Description automatically generated**

Figure 10 – example architecture of a simple CPU showing the fundamental registers

Overall, there are 7 steps to designing a CPU:

1. **Design specification:** involves the desired ISA (instruction mnemonics, codes and operation), width of the memory bus and the internal system bus and internal registers (e.g. GPRs or fundamental registers such as the Accumulator, AC, or the Instruction Register, IR)
2. **CPU states/execution routine design:** an execution routine/state takes one clock cycle to perform, and there may be N of them in a CPU operation. For example, the FETCH operation may commonly have 3 execution routines hence taking 3 clock cycles to fetch an instruction from memory). Execution routines need to be designed for each operation/instruction in the ISA.
3. **Execution routine optimisation:** reduction in the number of states in a single operation is crucial to increasing the efficiency of a CPU (measured in instructions/clock cycle)
4. **Datapath draft:** an initial block diagram of all the CPU internals wired up.
5. **Datapath optimisation:** a final block diagram optimising what needs access to the system bus and the widths of the internal busses.
6. **Arithmetic and Logic Unit (ALU) design:** a combinational component taking in two operands as its inputs with an internal MUX choosing which operation’s result to output (e.g. AND or ADD or OR)
7. **Control Unit (CU) design:** the “brain” of the CPU. It is the most important component in the CPU, it is what generates the control signals (what goes high or low when) according to which state the CPU is in. It generally consists of 3 parts:
   * ***State Counter:*** *holds current CPU state as a numerical value*
   * ***State Decoder:*** *takes its input from the counter, and holds the current CPU state as a classical decoder output (an N-bit decoder has only one of its outputs)*
   * ***Combinational Logic:*** *takes its input from the decoder and generates the correct state signals accordingly and control the state counter*

A black background with white text

Description automatically generated

Figure 11 – example of the FETCH execution routines; 3 clock cycles for the FETCH operation

**7 Design**

As mentioned before, the design process involved constructing all of the peripherals first, and then building the CPU at the end, to tailor the ISA and its architecture around the project’s needs. The design process outlined below is mentioned in chronological order.

**7.1 The VGA Controller**

**7.1.1** **The Pixel Clock**

The desired output resolution and refresh rate is 640x480 @60fps, and according to figure 2, a pixel clock of 25.175MHz is required. However, from research it was found that 25MHz would suffice. According to its manual, the Terasic DE0-CV development board has an onboard 50MHz crystal oscillator that may be as the clock. To obtain the desired 25MHz clock frequency, multiple approaches may be used but the simplest is to use a toggle flip-flop, whose output toggles at a frequency that is exactly half the input frequency used as its clock.

A black and white drawing of a square with a letter f

Description automatically generated

Figure 12 – T flip-flop block diagram. *Note: the input, T, is to be held constantly high (at Vcc)*

**7.1.2 HSYNC and VSYNC signals**

These two synchronisation signals will be generated using up counters. The reader is urged to review figure 1 to recall the VGA protocol.

1. HSYNC: the output of an up counter that counts from 0 to 799 (800 columns in total). HSYNC is held high if the counter < 656 and if 751 < counter < 801, otherwise the signal is low. This counter increments at each rising edge of the pixel clock.
2. VSYNC: the output of an up counter that counts from 0 to 524 (525 rows in total). VSYNC is held high if the counter < 490 and if 491 < counter < 526, otherwise the signal is low. **This counter only increments if the HSYNC counter = 800.**

The pseudocode example and diagram below may make it easier to understand.

**if(hsync\_cnt < ‘d656 && (hsync\_cnt > ‘d751 && hsync\_cnt < ‘d801)) begin**

**...**

**hsync <= 1;**

**end**

**else begin**

**...**

**hsync <= 0;**

**end**

**A diagram of a diagram

Description automatically generated**

Figure 12 – block diagram of synchronisation signals. The “row” and “column” are the real-time coordinate of the scanline. The “rgb\_en” outputs will be connected to the RGB controller (see the next section)

**7.1.3 RGB Controller**

This device is what allows colour to be displayed. According to the protocol, colour may only be displayed in the active region. So, if the column > 639 or the row > 479 the output RGB output must be zero. To accomplish this, the device will have an active high input “enable” pin that will be the XNOR of the two “rgb\_en” outputs in figure 12, such that when either of them go low, the enable pin goes low as well, and the RGB outputs are off. This module also needs to accept 12-bit colour data as well. Where this data comes from will discussed in future sections.

A drawing of a diagram

Description automatically generated

Figure 13 – block diagram of the RGB controller

**7.1.4 ROM Sprites**

A ROM sprite is a pre-defined graphical element stored in Read-Only Memory, commonly used in old game consoles to represent characters, objects, or backgrounds within video games. *[chat.openai.com].* Every game entity or object would have their coordinates and colour data stored in a ROM and would only be displayed when the scanline matches the stored coordinate for that pixel. This is what will be referred to as “hardware rendering”, since all the information (coordinates and colour data) is stored in silicon, rather than being controlled by a CPU. “Software rendering” by the CPU can be done, but it will be explained later why this is not a good idea.

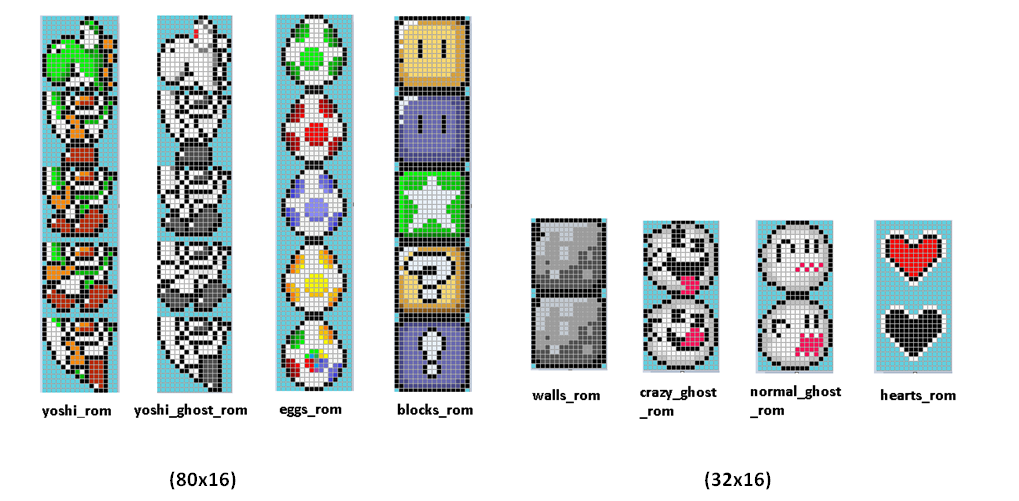


Figure 13 – example of sprites. Each square represents a pixel in memory [12]

**For an N-pixel sprite, an N-bit ROM will be required with an N number of if statements for each pixel.** Consider the 4-pixel sprite below.

A red square with four squares

Description automatically generated

Figure 14 – 4-pixel square sprite. *This sprite was made by* [*http://www.piskelapp.com*](http://www.piskelapp.com)

To store this sprite, a ROM with 4 x 12-bit memory locations would be required.

**7.1.4 Movement of Sprites from User Input**

Thus far, the sprites that would be rendered are static and cannot move. To allow for input-controlled movement, the idea of “pixel offsets” was used where, rather than rendering a pixel at a specific coordinate, the RTL if-statement would contain an offset value contained within internal registers. Assuming only 4-inputs are allowed (up, down, left and right), 2 internal registers would be used to keep track of row offsets and column offsets. These registers are simply counters that would be incremented/decremented accordingly. For example, suppose one of the registers is called “offset\_row” which would hold the pixel offset responsible for horizontal movement of the sprite. Rather than doing the following:

if(row == ‘d250) begin

...

// display pixel RGB data

end

this would be done instead.

if(row == (‘d250 + offset\_row)) begin

...

// display pixel RGB data

end

There will be 4-bit register that stores user input. To keep it simple, only 1 input key is allowed at a given time. By default, no keys are pressed, and the user input register would store 4’b0000. If a certain key is pressed to move, say, the sprite to the right, the register would store 4’b1000, and the offset\_row register would be decremented to move the sprite to the left. It is important to note that these registers need to be of type signed to allow for negative numbers, so one extra bit will be needed for the sign bit.

A drawing of a wiring diagram

Description automatically generated

Figure 15 – sketch of proposed idea

Where r1 and r2 in the Pixel Offset Counter module are the two internal registers offset\_row and offset\_column, and r3 and r4 in the Sprite ROM store these values and are used in the RTL if-statement for rendering.

**7.1.5 Pixel Edge Detection**

If the design above were to be implemented, the sprites would move but can move off the screen as well. To prevent this, the current position of the pixels would need to be known by the Pixel Offset Counter module, such as to prevent the increment of the offset registers if the pixels are at the edge.

The initial idea was to store the value of all the pixels in the sprite and go off from there. However, that was hugely unnecessary. Instead, a more optimised design requiring the current position of only 4 pixels was used. For any sized sprite, if the vertex of each of the 4 sides facing the edges of the screen where to be taken, it would suffice for the edge detection mechanism. See the figures below for guidance.

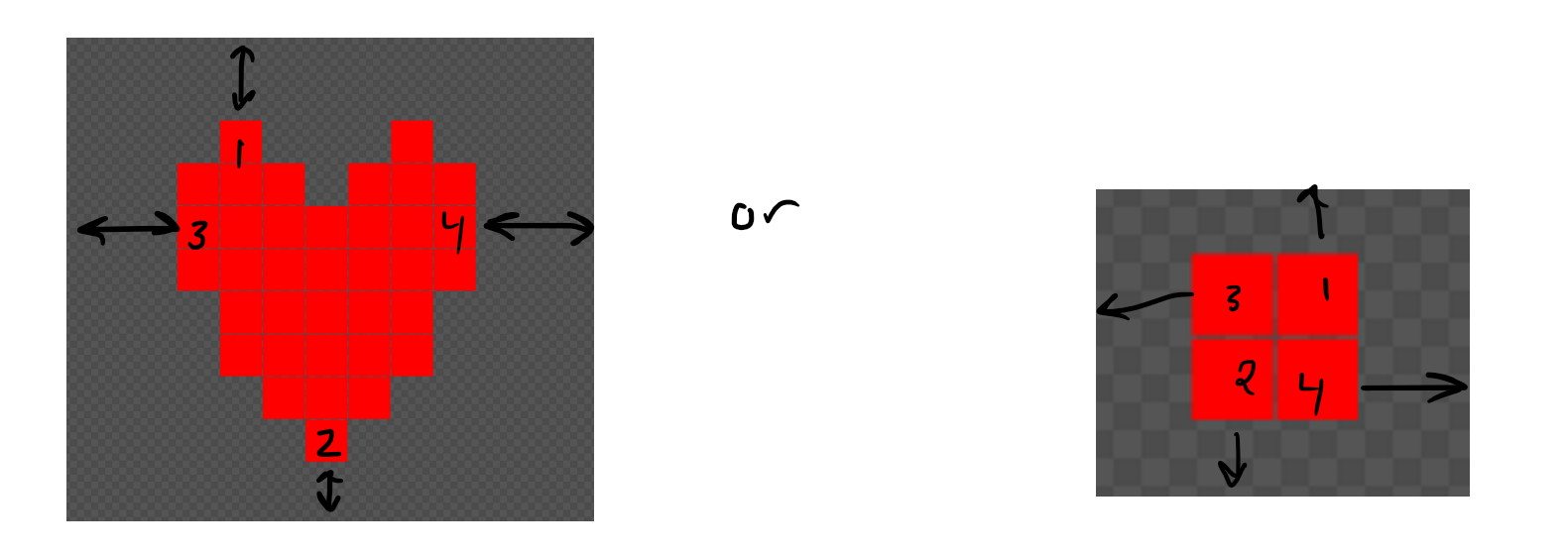


Figure 16 – example sprites with edge detection of 4 “vertex pixels”. *These sprites were made by* [*http://www.piskelapp.com*](http://www.piskelapp.com)

4 registers internal to the sprite ROM would be used to store the current row positions of pixels 1 and 2, and the current column positions of pixels 3 and 4. For example, using the square sprite on the right (assuming that pixels 1, 2 were to be rendered at rows 100 and 101, and pixels 3 and 4 at columns 100 and 101):

Pixel1\_row\_pos = 100 + offset\_row;

Pixel2\_row\_pos = 101 + offset\_row;

Pixel3\_column\_pos = 100 + offset\_column;

Pixel4\_column\_pos = 101 + offset\_column;

Next, these registers would be broken out of the ROM and into the Pixel Offset Controller module as inputs like the sketch below.

A drawing of a bus

Description automatically generated

Figure 17 – sketch showing how the 4-pixel coordinate data would be wired

Using this data for edge detection would be straightforward: do not increment/decrement the offset counters any further if the pixel in question is at the edge. For example.

if(pixel1\_column\_pos == ‘d639 || pixel2\_column\_pos == ‘d0) begin

...

// do not increment/decrement offset registers

end

**7.2 The PS/2 Controller**