**Designing and Implementing a 32-bit FPGA Computer**

By

Youssef Elgedawy

A Dissertation

Submitted to The School of Engineering, Computing and Mathematics (SECAM)

University of Plymouth

In Partial Fulfilment of the Requirements For the Degree of

Bachelor of Electrical and Electronics Engineering

**May 2024**

**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 were 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.

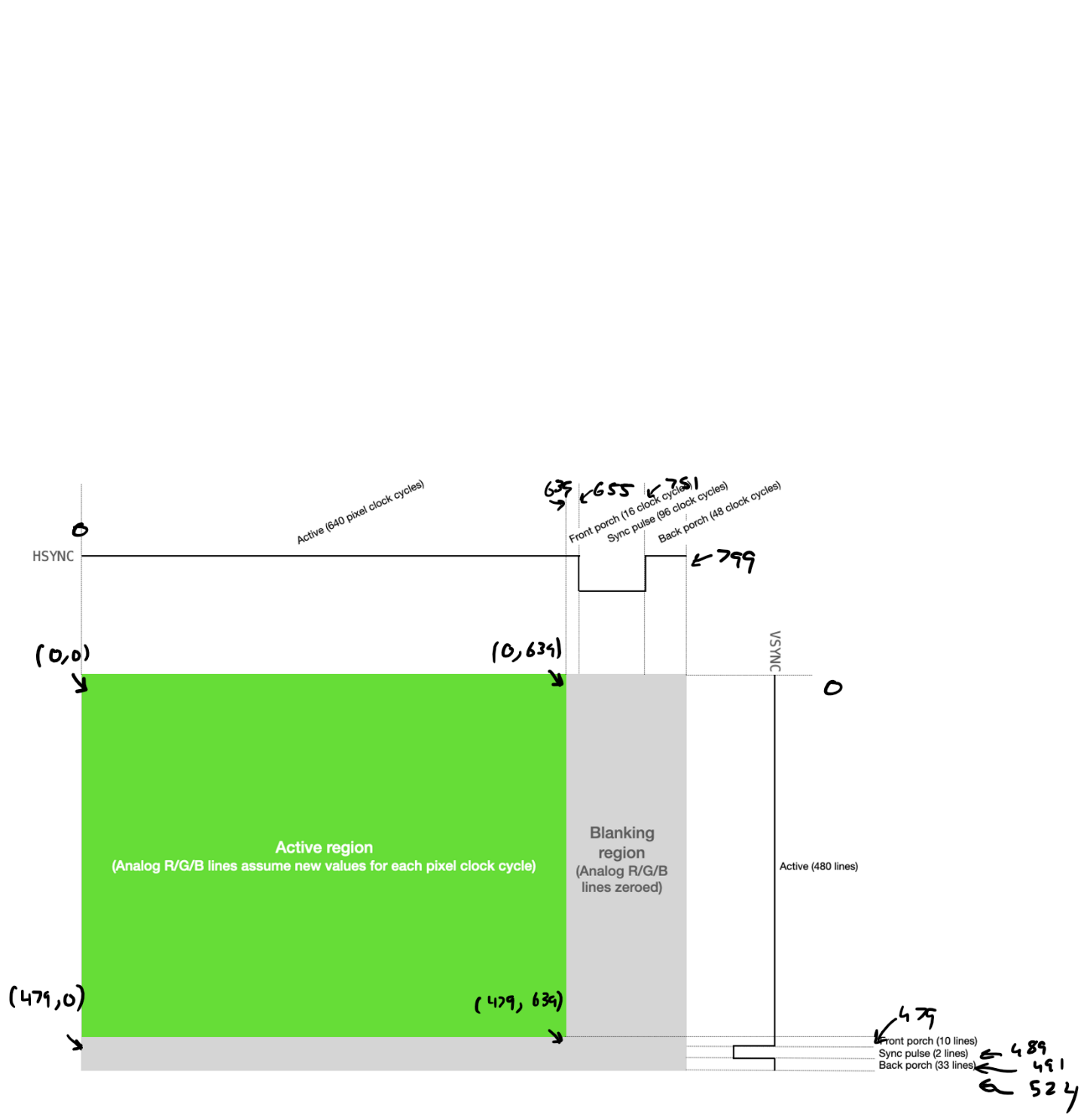


Figure 1 – VGA protocol diagram for a 640x480 resolution [1] with the authors own annotations where the numbers in brackets represent *(rows, columns).*

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.

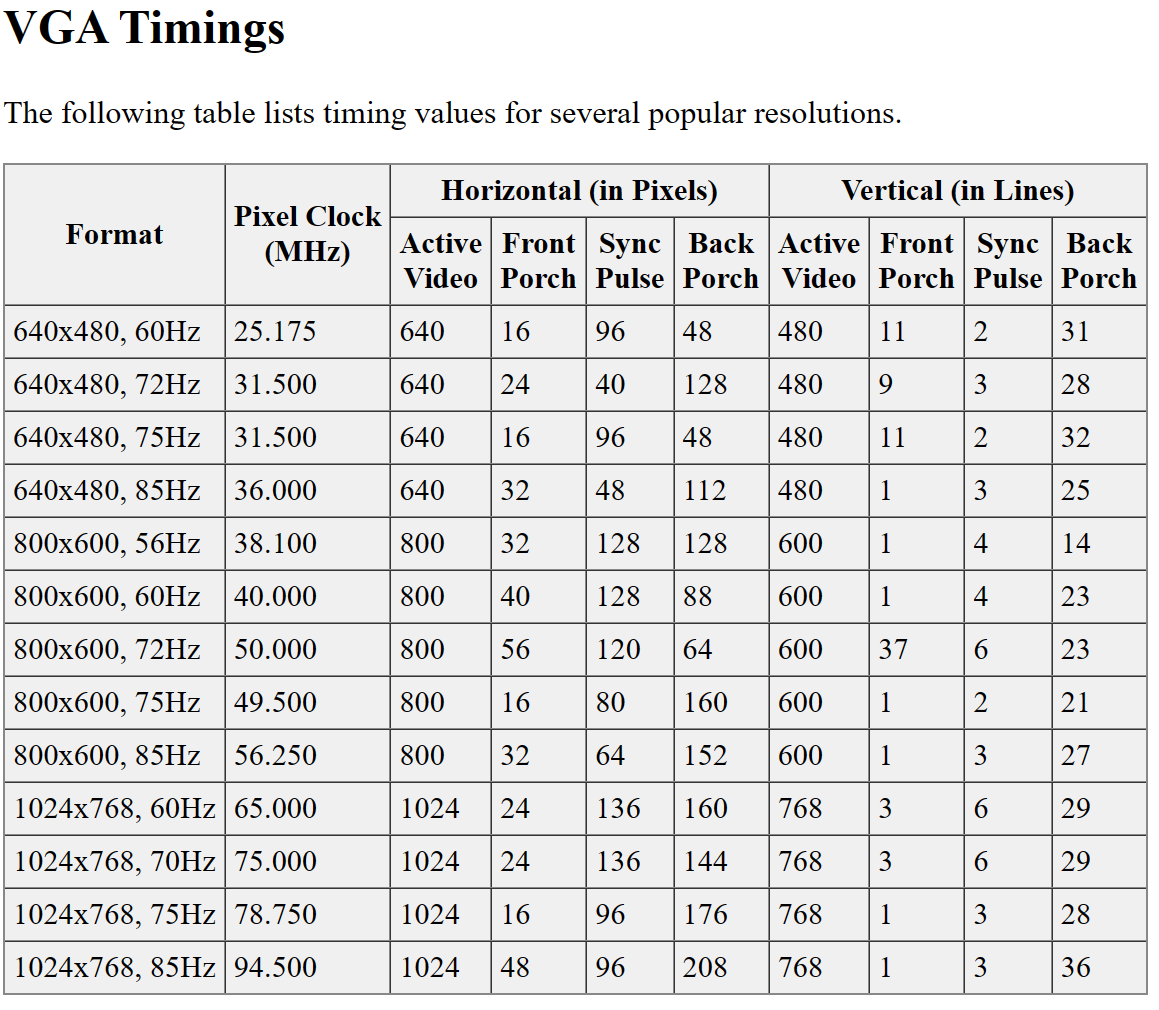


Figure 2 – table of pixel clock frequencies according to resolution and refresh rate [2] *it may be noted that slight deviations of those frequencies will still satisfy the protocol implementation.*

The key parts of the protocol are as follows:

1. **Active region** – the region in the display that the viewer sees the image in (in this case it is 640 columns by 480 rows).
2. **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.
3. 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 is shown below.

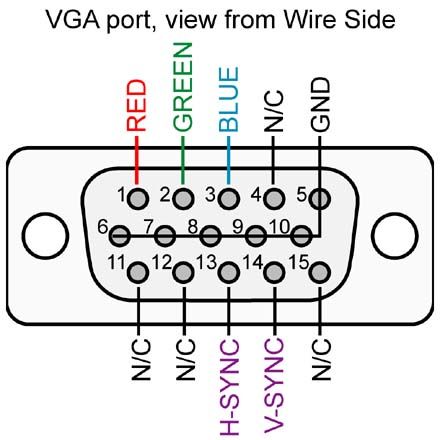


Figure 4 – DB-15 pinout [5].

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.

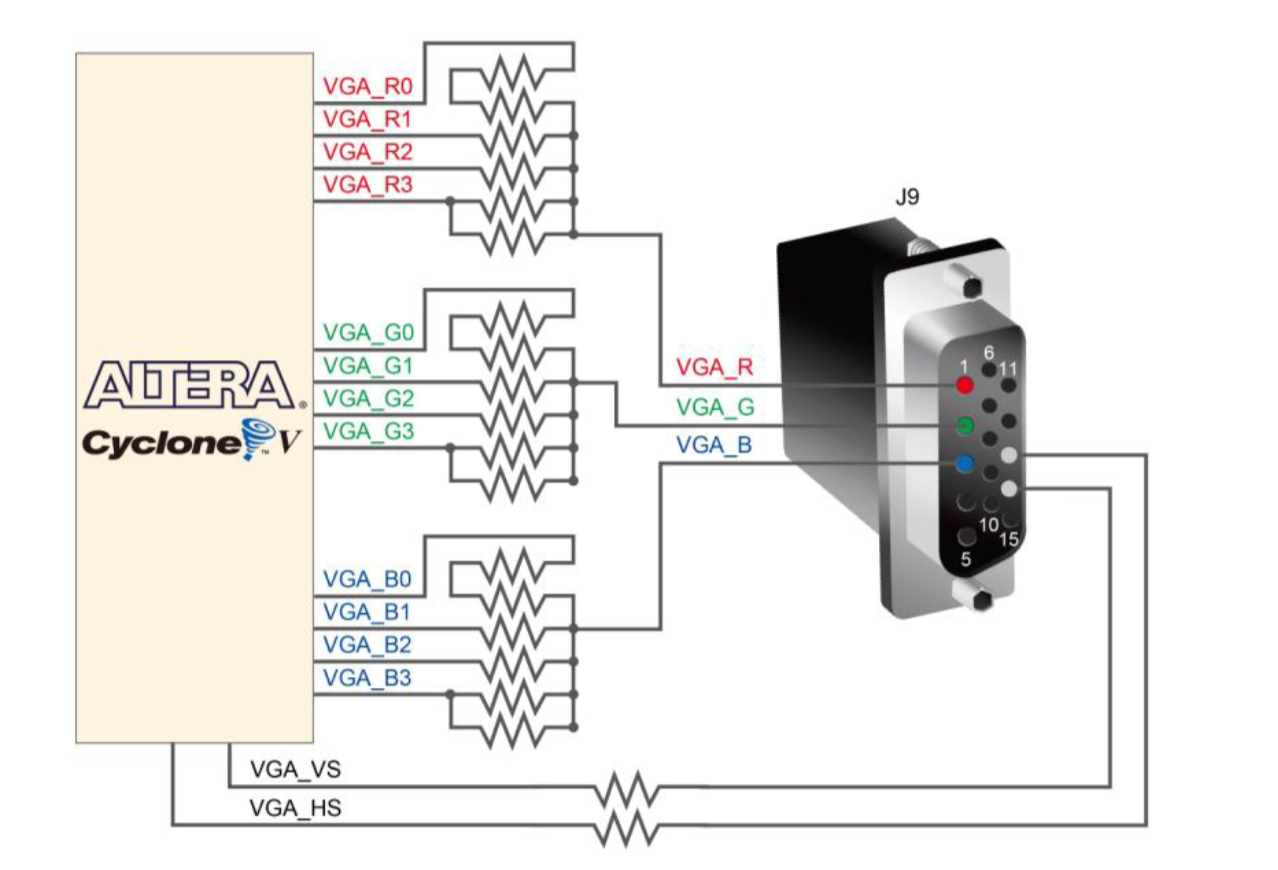


Figure 5 – DB-15 pinout on the DE0-CV board, taken from its user manual [8].

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:

1. **Agreement on baud rate (data rate in bits/second – bps):** both the Tx and the Rx must know what baud rate to communicate with.
2. **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.

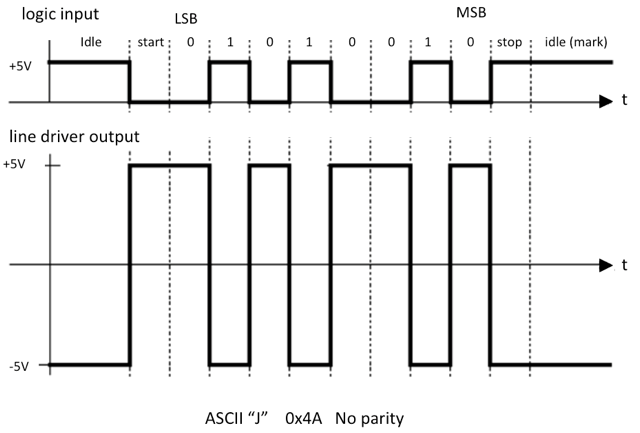


Figure 6 – RS232 frame structure. Top: logic input (and output if no line driver is used). Bottom: line driver output [9].

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.*

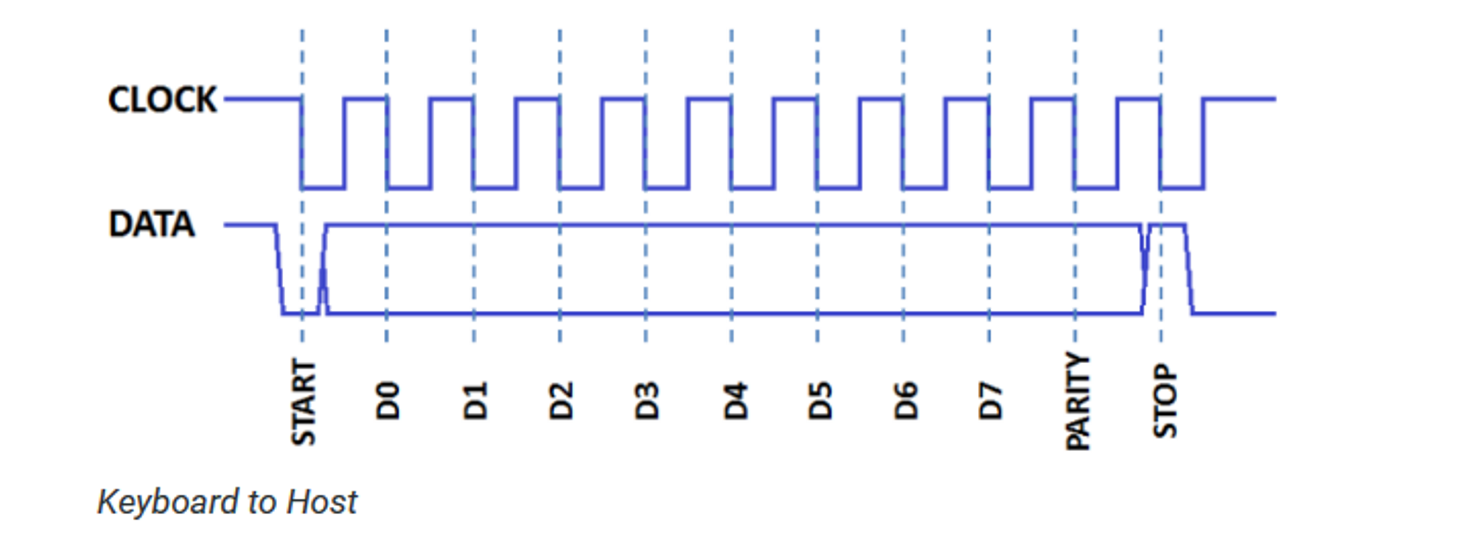


Figure 7 – PS/2 keyboard frame structure [10].

**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.

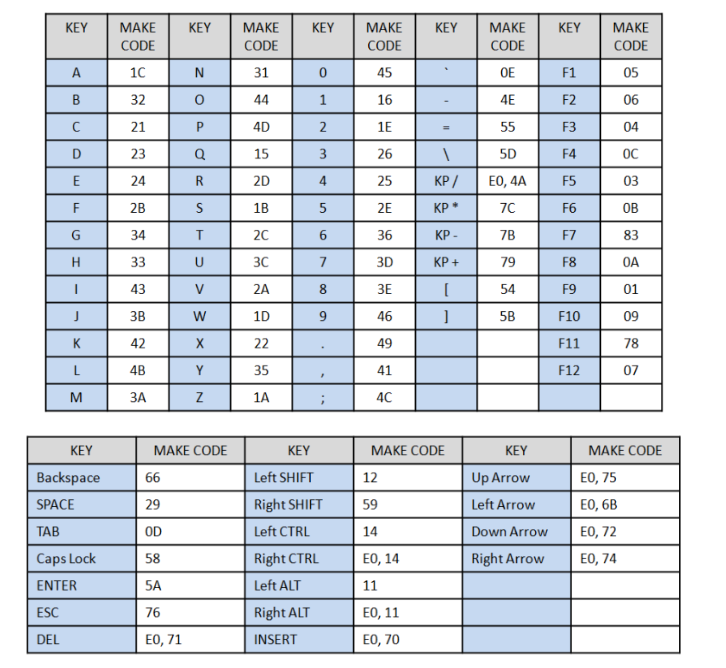


Figure 8 – PS/2 keyboard scan codes [11].

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.

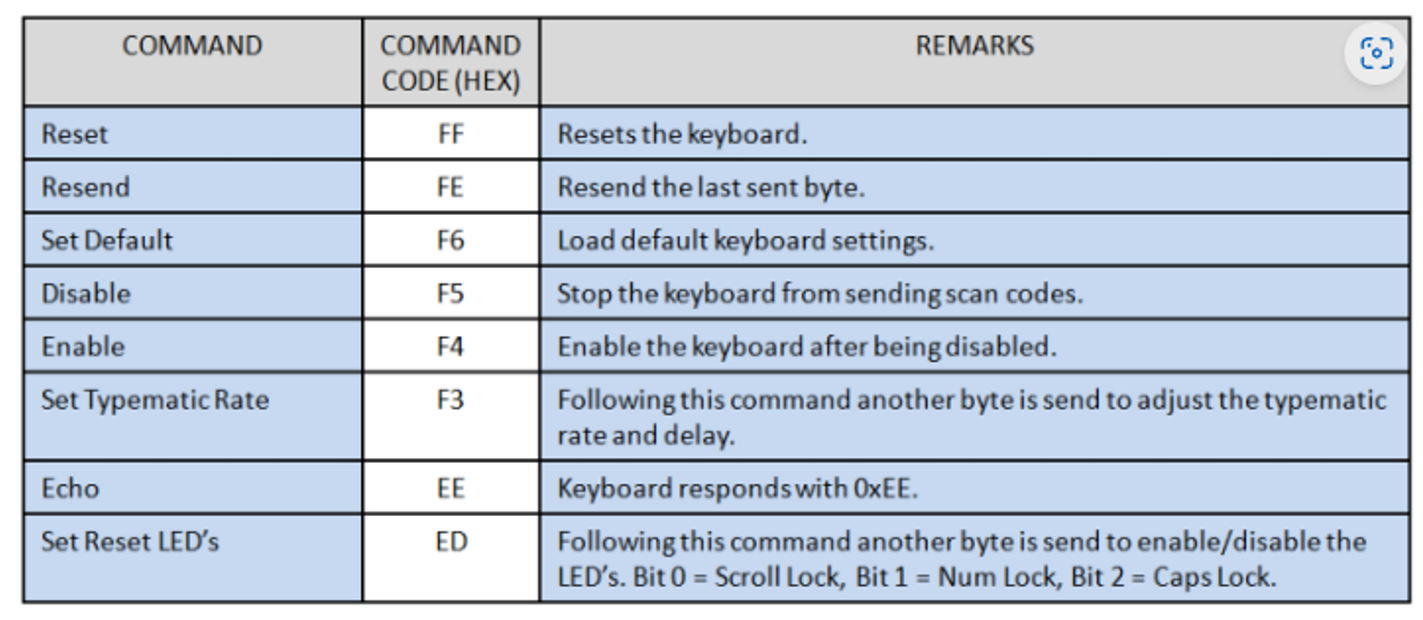


Figure 9 – PS/2 keyboard hex commands sent from “device” to “host” [11].

**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.

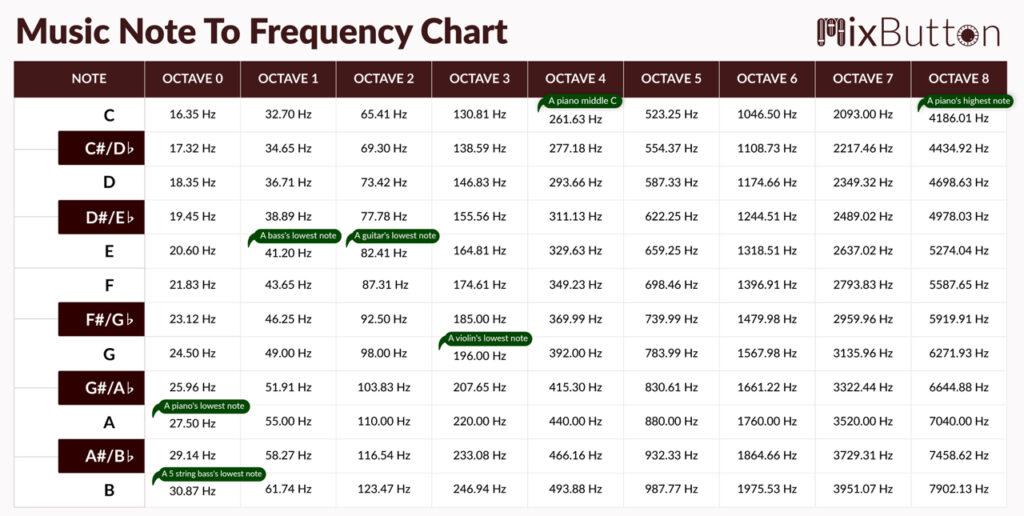


Figure 9 – musical notes frequency mapping [12].

**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:

1. **Fetch cycle:** fetch an instruction from memory, then go to the decode cycle.
2. **Decode cycle:** decode the instruction—that is, determine which instruction has been fetched—then go to the execute cycle for that instruction.
3. **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:

1. **Address Register (AR):** stores the address of the current instruction to be fetched from memory.
2. **Program Counter (PC):** stores the address of the next instruction to be fetched from memory.
3. **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.
4. **Accumulator (AC):** stores the result of any arithmetic and logic operation on operands (output from the ALU).
5. **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 seven 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. **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)
5. **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.*
6. **Datapath draft:** an initial block diagram of all the CPU internals wired up.
7. **Datapath optimisation:** a final block diagram optimising what needs access to the system bus and the widths of the internal busses.

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 be 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.

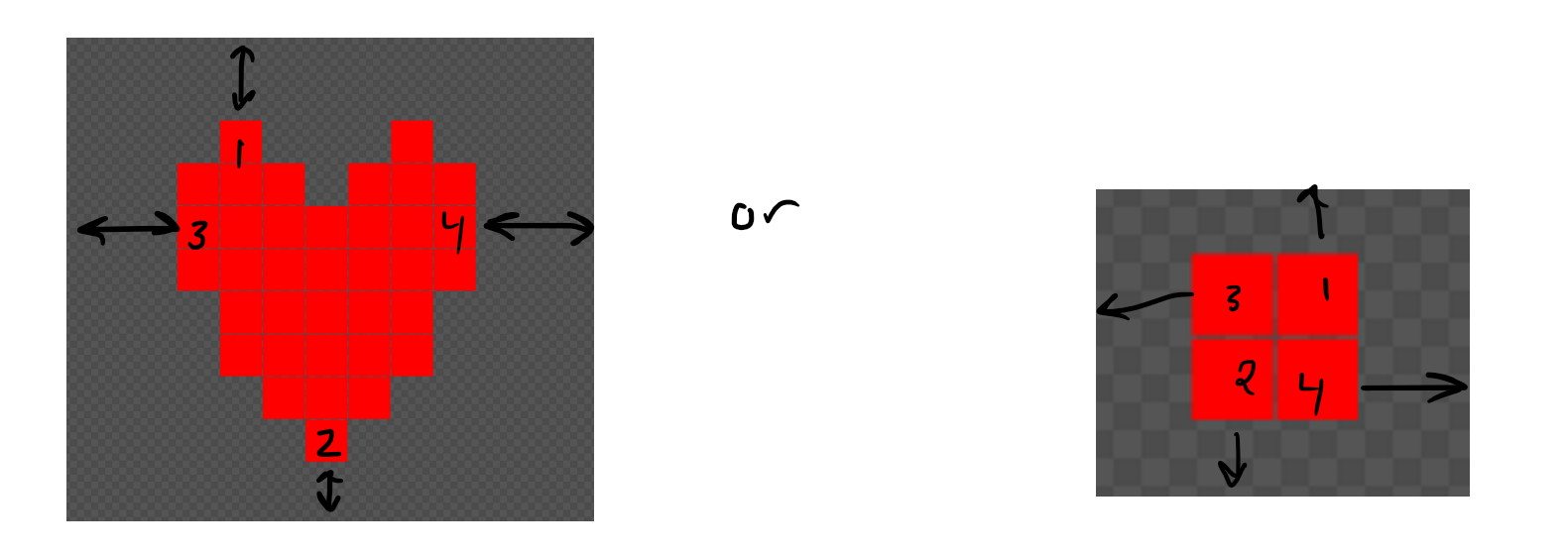


Figure 16 – example sprites with edge detection of 4 “vertex pixels”. *These sprites were made by* [*http://www.piskelapp.com*](http://www.piskelapp.com)

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 (Keyboard)**

As the host (keyboard) only generates a clock when a key is pressed meaning that the clock line is kept high when idle, a simple counter may be used to help implement the controller.

The following steps were used to implement the protocol:

1. Controller must know that the start bit is low, and the stop bit is high.
2. According to the protocol, when the clock starts, it must sample the received bits on the falling edge of the clock.
3. An internal 4-bit counter register will be used to keep track of the number of falling edges detected; each falling edge 🡪 count++.
   * count = 0; idle.
   * count = 1; start bit.
   * 2 ≤ count ≤ 9; data frame – this is the only time the data is placed inside a SPSR for storage.
   * count = 10; odd parity bit.
   * count = 11; stop bit – use this as a reset back to count = 0 to signify eot (end of transmission).
4. An internal SPSR (serial-parallel shift register) will be required.
5. A busy flag will be used as follows:
   * busy = 0 when count = 0.
   * busy = 1 for entire frame.
   * then busy = 0, and data may be read off the SPSR.

A diagram of a computer program

Description automatically generated with medium confidence

Figure 18 – block diagram showing host and FPGA-based PS/2 interface.

**7.3 The UART**

**7.3.1 The Receiver**

The sole purpose of the UART is to act as a communication interface between any UART capable device (such as a MCU) and the FPGA in order to later on program the CPU by sending instructions to an instruction ROM. As mentioned in the abstract, only the receiver portion of the UART will be implemented as the CPU will not need to transmit anything back.

Due to the similarity between the PS/2 and the RS232 protocols, and with the PS/2 controller planned to be fully implemented at this point, the same design approach of using a counter was going to be used. However, one major difference between the two protocols is that the RS232 protocol is asynchronous, and the counter method requires a clock to count its rising/falling edges.

To combat this issue a sampling clock had to be generated. The 50MHz clock was to be divided down to suitable frequency based on the agreed desired baud rate. The following explanation assumes a baud rate of 300 bps but can easily be extended with any required baud rate.

First, the time period of the sampling clock must be determined.

More importantly, another value known as the bit period is needed where,

This is the total time the bit data is going to be asserted for.

A drawing of a diagram

Description automatically generated

Figure 19 – diagram showing time period (6.6ms) and bit period (3.3ms).

For reliable sampling, the sampling edge of the clock must line up with half of the bit period (1.65ms in this case) and away from the edges.

*(see next page)*

A drawing of a square with arrows

Description automatically generated with medium confidence

Figure 20 – Top: example UART data at 300 baud. Bottom: sampling clock with time period = half bit period.

Thus, the sampling frequency is:

With this important relationship derived, another problem needs to be tackled: this clock must only be generated when a frame is being set. This can be achieved by having the sampling clock module sense the start bit (low on the serial data line) and by sensing the positive edge of the busy flag – indicating eot.

The final problem that needs to be addressed is the fact that the sampling clock needs to start in the high 🡪 low state to meet timing requirements.

A diagram of a computer hardware system

Description automatically generated

Figure 21 – connection diagram between RS232 receiver and the sampling clock.

A screen shot of a graph

Description automatically generated

Figure 22 – oscilloscope waveforms. Blue: serial data. Red: raw sampling clock.

A screen shot of a graph

Description automatically generated

Figure 23 – desired oscilloscope waveforms. Blue: serial data. Red: sampling clock active during frame. Green: busy flag.

Needless to say, the UART module also has an internal SPSR that may be read on the rising edge of the busy flag.

*It should be noted that as an alternative to manually creating a sampling clock, an external clock could be used if the Tx has a USART (not a UART). But, due to this being not very common and the desire to have a minimal programming interface (a single wire), this approach was disregarded.*

**7.3.2 Writing to Memory**

This transmitted data must be stored somewhere for later use – a synchronous RAM. This RAM will have 3 modes:

1. **Write** –to instruction memory.
2. **Debug** – the use of seven segment displays to display the contents stored.
3. **Random access fetch** – by the CPU.

A few considerations needed to be taken care of.

**Write mode:** how will the RAM know when to increment its internal address counter to write in its next memory location? Two solutions were proposed:

1. Sending and detecting a special character (e.g. ASCII ‘$’)
2. The detection of an eot flag (optimal)

The first method was initially used and then the second method was adopted instead (see the implementation section for more details).

**Debug mode:** the use of a push-button to increment the internal address counter to display the contents found in the RAM is a feasible idea. However, to make debugging reliable, button bounce had to be accounted for. The use of a hardware debouncer module (timer) whose output follows the input after a delay was used.

*(see next page)*

A diagram of a square

Description automatically generated

Figure 24 – Instruction RAM connections.

**7.3.3 Intermediate Analogue Circuitry**

*This section will discuss voltage levels, line drivers and interfacing with TTL/CMOS devices, and is only relevant if the UART capable device being used makes use of a line driver to output the serial data.*

Traditionally, this serial data was sent to a DB-25 connector serial port (AKA RS232 port) on a computer. The RS232 protocol voltage levels defined the following voltage levels (note the logical inversion):

1. Logic level low: +5V 🡪 +15V
2. Logic level high: -5V 🡪 -15V
3. Undefined: -5V < voltage < +5V – interpreted as noise

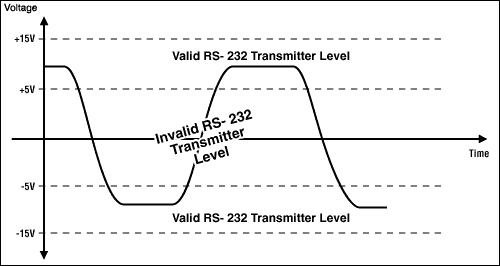


Figure 25 – RS232 voltage levels [13].

Nowadays, the devices around us use 5V or 3.3V logic levels (either TTL or CMOS), so a UART implemented using these voltage levels cannot be connected directly to the RS232 port – completely different voltage levels. In order to convert the 5V or 3.3V to the RS232 voltage levels, a device known as a Line Driver is used. **If it is used, care must be taken so as not to damage the FPGA I/O pins** since according to the *Intel Cyclone V device datasheet [14]* under the *“Absolute Maximum Ratings for Cyclone V Devices”* section, the *maximum DC input voltage is 3.80V.*

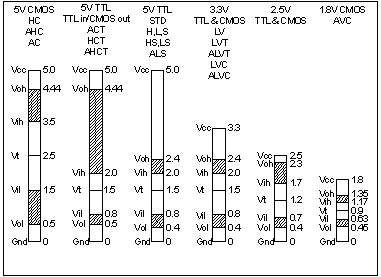


Figure 26 – TTL and CMOS voltage levels [15].

So, to interface the different voltage levels together, the following example circuitry may be used (assuming -5V 🡪 +5V output voltages).

A diagram of a circuit

Description automatically generated

Figure 25 – block diagram of interface circuitry.

A diagram of a circuit

Description automatically generated

Figure 26 – schematic design of figure 25.

A screen shot of a computer

Description automatically generated

Figure 27 – Tx to FPGA waveforms. Green: line driver output. Red: converted FPGA input.

A screen shot of a computer

Description automatically generated

Figure 28 – Rx from FPGA waveforms. Green: FPGA output. Red: converted line driver input.

**7.4 PWM Audio Controller**

The choice of audio output for this project is a simple PWM buzzer that can be driven by an FPGA I/O pin. Looking back at figure 9, the ability to choose one of many different audible frequencies is needed. To achieve this, an array of very accurate frequency dividers will be created, and, through the use of a MUX, a single frequency will be output to the buzzer.

A diagram of a computer program

Description automatically generated

Figure 29 – example of a 4-frequency divider array.

The above design can easily be scaled to accommodate more frequencies. Focusing only on octave IV, 13 different frequencies are required (12 frequencies + 0Hz; off). Thus, a 16x4 MUX may be used. But what will be connected to the select bus? Initially, it was decided to hard-wire the PS/2 keyboard output to a key detector module which outputs a MUX select value according to the key being pressed.

A diagram of a key detector

Description automatically generated

Figure 30 – design incorporating the 13 musical notes.

Each musical note in figure 30 correspond to a frequency divider module connected in the order above (i.e. sel = ‘d0 🡪 buzzer\_out = off; sel = ‘d1 🡪 buzzer\_out = B etc…). The Key Detector module will be a combinational component implementing the following lookup table:

|  |  |
| --- | --- |
| **PS/2 data input** | **Select output** |
| B | 1 |
| A# | 2 |
| … | … |
| C# | 11 |
| C | 12 |
| default (off) | 0 |

Figure 31 – Key Detector lookup table.

To implement the sharp keys, the input combination ENTER+key will be used e.g. A# will be played when ENTER+A is pressed. Since only one key can be detected at a time, if the ENTER make code (0x5A) is detected, the audio controller will enter a state in which it waits for another key to be pressed – so the ENTER can be thought of as a “sharp toggle”.

**7.5 The CPU**

**7.5.1 Design Specification**

This section is considered the largest and most important in the entire project and care had to be taken in constructing the CPU’s capabilities to be sufficient to interface with all the peripherals with ease.

**7.5.1.1 The Instruction Set Architecture**

As was briefly outlined in section 6.5, there are (generally) sevens steps to follow in order to design a CPU. These steps are only meant to be an aid.

Prior to planning, inspiration was taken from the ARMv7 ISA [16]. So, any similarities in architecture are not accidental.

This CPU was planned to have a 16 of the most commonly used instructions in programming today, meaning that a 4-bit opcode is necessary. So, the following ISA was designed.

A table with numbers and symbols

Description automatically generated

Figure 32 – CPU ISA, separated into general instructions (GI) and arithmetic and logic instructions (ALI).

A screenshot of a computer

Description automatically generated

Figure 33 – ISA details.

Where:

1. **a** and **b:** operand bits.
2. **s:** selector bits.
3. **x:** don’t care bits.

A big challenge when designing this ISA was thinking about how to incorporate two different versions of the same instruction into one word (38-bits). Consider the mov instruction for example, how can the programmer choose between moving an immediate/literal (constant) and the contents of another register?

One approach was to make separate them into two different instructions with different mnemonics and opcodes (e.g. movi and movr for immediate and register moves respectively). However, it was promptly realised that this would make the ISA very “bloated” and would require more than 4-bits defining the opcode. Thus, another approach was needed.

Alternatively, the instruction word was designed to incorporate 2 special bits known as *selector bits.* These bits would be used to choose between the different modes of an instruction. The use of 2-bits implies that up to 4 different modes of a single instruction can be designed and selected. However, for the GI, only the lower selector bit s[0] was used to select between a register operand and an immediate operand. It was then decided that if that bit was low, the operand would be an immediate, and if the bit was high, the operand would be the contents of another register.

This ISA uses 16-bit operands (a and b). These operands bits are used differently depending on the combination of select bits. Consider the operation of the mov instruction.

**reg( a[15..12] ) <- b / reg( b[15..12] )**

**LHS:** The top 4 bits of the operand a bits are used to select the destination register (a choice of up to 16 registers).

**RHS (s[0] == 0) – immediate operand is selected:** the 16-bits of operand b are used.

**RHS(s[0] == 1) – register operand is selected:** only the top 4 bits of the operand b bits are used to select the source register.

For example, if s[0] == 0; a[15..12] == ‘d0 and b = ‘d15, this would imply the following instruction

**mov r0, #15**

The disadvantage of this method however is that, for most of the ALI, the operand bits are only 12-bits instead of 16 due to squeezing in 4 different modes of a single instruction into a single 38-bit instruction word. Consider the different modes of the add instruction below:

**Mode 1:** add r0, #5, #6 // r0 = 5 + 6

**Mode 2:** add r0, #5, r1 // r0 = 5 + r1

**Mode 3:** add r0, r1, #5 // r0 = r1 + 5

**Mode 4**: add r0, r1, r2 // r0 = r1 + r2

**7.5.1.2 The Internal Registers**

Each internal CPU register along with its function is described below.

1. **13 general purpose registers (r0 – r12):** *32-bits wide;* used for fast temporary data storage.
2. **2 operand registers: (rop1, rop2):** *16-bits wide;* used to store the operand bits, a, and b, for later use.
3. **1 selector register: (rsel):** *2-bits wide;* used to store the two selector bits, s[1] and s[0].
4. **1 temporary register (TR):** *32-bits wide;* used for temporary data storage during some CPU execution routines.
5. **CPU flags registers (CPU\_flags):** *4-bits wide;* used to hold the NZCV flags.
6. **Address register (AR):** 12-bits wide; holds the address of the current instruction to be fetched from memory. 4096 addresses split between instructions, stack, and memory mapped registers.
7. **Program counter (PC):** *12-bits wide;* holds the address of the next instruction to be fetched from memory.
8. **Data register (DR):** *38-bits wide;* holds the instruction word from memory.
9. **Accumulator (AC):** *32-bits wide;* holds the ALU result. The width of this register is exactly double the width of the operand bits to accommodate for the multiply instruction without overflowing.
10. **Instruction register (IR):** *4-bits wide;* holds the opcode portion of the instruction word.

A group of rectangular objects with writing on them

Description automatically generated

Figure 34 – CPU internal registers, along with their widths.

**7.5.2 Draft CPU Execution Routines**

Again, a CPU execution routine/state involves the movement of data between its internal registers **in a single clock cycle.** Care must be taken to minimise the number of unnecessary steps as this will slow down the overall rate of instruction execution by the CPU. The CPU has 17 operations (16 instructions + FETCH operation), and the execution routines for each will be discussed below.

1. **Fetch (FETCH) – 3 clock cycles**

**A close-up of a number

Description automatically generated**

Figure 34 – the FETCH operation. ‘M’ stands for memory.

Notice how in state fetch3, 4 registers were written to in a single clock cycle/state. This is possible as long as the following rule is never broken:

***A register cannot hold the updated value of another register if the update happened in the same state.***

For example, the follow is **not** valid if the designer intended for reg2 to hold the updated value of reg1 due to clock cycle delays; reg1 will be updated with value on the next positive edge (next clock cycle), but reg2 will hold the previous value of reg1.

A black and white text

Description automatically generated

Figure 35 – example of an illegal state.

The minimum number of clock cycles the whole operation can take is 2:

A group of black text

Description automatically generated

Figure 35 – example state now legal!

1. **No Operation (NOP) – 1 clock cycle**

A close up of a word

Description automatically generated

Figure 36 – the NOP operation.

This instruction is usually used for timing synchronisation if required.

1. **Move (MOV) – 1/2 clock cycles**

A white background with black text

Description automatically generated

Figure 36 – the MOV operation, where ALT stands for alternative.

This instruction allows the movement of data into a register. This data can either be a literal or the contents of another register. The operand can be up to 16-bits without the need for a separate instruction (like the MOVW instruction in ARM assembly)

Notice the two conditional paths this instruction may take depending on the value of the lower selector bit. Despite the two different execution paths, the same opcode is sent – these are not two different instructions. The different mnemonics are just used for clarity.

1. **Load Register (LDR) – 2/4 clock cycles**

**A white background with black dots

Description automatically generated**

Figure 37 – the LDR operation.

This instruction either loads a register with the address of data found in memory, or its actual (dereferenced) value.

1. **Store Register (STR) – 4/4 clock cycles**

**A screenshot of a computer

Description automatically generated**

Figure 38 – the STR operation.

This instruction either stores the contents of a register, or a literal into memory. The latter was used to implement variable support in the assembler later.

1. **Compare (CMP) – sets NZCV; 1 clock cycle**

****

Figure 39 – the CMP operation. ‘r’ is any GPR.

This is a special form of the sub instruction where rsel[1] = 1 to set the first operand as r, and data (second operand) depends on rsel[0] (literal/register).

1. **Unconditional Branch (B) – 1 clock cycle**

****

Figure 40 – the B operation. *Also known as BAL in ARM assembly.*

This instruction is similar to the goto instruction in C; upon execution of this instruction, the CPU will always unconditionally jump to whatever address is specified.

1. **Branch If Greater Than (BGT) – 1 clock cycle**

****

Figure 41 – the BGT operation.

This is the first of 3 conditional branch instructions. These instructions are meant to be used right after the CMP instruction due to it setting flags. The conditional branch instructions check for a particular pattern in the NZCV flags that, if met, will satisfy the condition of the branch. The implementation of these patterns were done by using the following table.

*A screenshot of a computer

Description automatically generated*

Figure 42 – *Table 5.1 in the ARM Cortex-R Series Programmer’s Guide [17].*

1. **Branch If Less Than (BLT) – 1 clock cycle**

**A black and white text

Description automatically generated**

Figure 44 – the BLT operation.

1. **Branch If Equal To (BEQ) – 1 clock cycle**



Figure 45 – the BEQ operation.

1. **Addition (ADD) – sets NZCV; 2 clock cycles**

**A close up of a text

Description automatically generated**

Figure 46 – the ADD operation.

Recall that data depends on the selector bits and r stands for any general-purpose register (GPR).

1. **Subtraction (SUB) – sets NZCV; 2 clock cycles**

**A close up of a word

Description automatically generated**

Figure 47 – the SUB operation.

1. **Multiply (MUL) – 2 clock cycles**

**A close up of a word

Description automatically generated**

Figure 48 – the MUL operation.

1. **Logical Shift to The Right (LSR) – 2 clock cycles**

**A group of black letters

Description automatically generated**

Figure 49 – the LSR operation.

This instruction was used as a means of adding division capabilities to the CPU in a very fast and inexpensive (little FPGA resources) way. Since lsr N is equivalent to dividing by .

1. **Logical AND (AND) – 2 clock cycles**

**A close up of text

Description automatically generated**

Figure 50 – the AND operation.

1. **Logical OR (OR) – 2 clock cycles**

**A close up of text

Description automatically generated**

Figure 51 – the OR operation.

**12. Move Logical Not (MVN) – 2 clock cycles**

**A group of black text

Description automatically generated**

Figure 52 – the MVN operation.

This instruction serves to move the inverse of the operand provided. For example (assuming 8-bit register widths)

**mvn r0, #0x00**

would move 0xFF into GPR r0.

This concludes the design of *all 40* CPU states.

**7.5.3 CPU Execution Routines Optimisation**

This stage of the design involves optimising the number of CPU states; removing redundant states and replacing them with a single state that achieves what is required. Two optimisation examples are shown below.

A white background with black text

Description automatically generated

Figure 53 – removal of redundant state.

When the select bit is low, the LDR operation is meant to load the address of data found in memory (e.g. a variable). When the select bit is high, the desired register is to hold the actual value of that data. Thus, state ALTldr2 is redundant and may be removed.

A screenshot of a computer

Description automatically generated

Figure 54 – removal of another redundant state.

When the select bit is low, the STR instruction stores GPR contents into memory. The use of the temporary register (TR) is unneeded here. Thus, state str2 is redundant and may be removed. Instead, str2 becomes,

str2 : DR <- r

With these two optimisations alone, the total number of CPU states could be *reduced from 40 down to 38.*

**7.5.4 Choosing Between the General-Purpose Registers (GPRs)**

Again, this CPU has 13 GPRs that can all be used at any time. But how will the hardware know which GPR to choose and what to do with it? A module known as the *GPR Selector* will be designed. Its architecture is outlined below.

A diagram of a circuit

Description automatically generated

Figure 55 – GPR Selector module; makes use of a DEMUX-MUX pair.

Referring back to the ISA (figure 32), some instructions have the capability of choosing which GPR register to work with using the top 4 bits of operands a and b. This design shows the reader how this is done. 4 bits are used for a choice of up to 16 GPRs. The outputs of operand registers rop1 and rop2 are hard-wired (directly connected to; not needing to access the CPU system bus) to this module’s select busses.

The load bus is the data to be loaded in one of those registers.

The second output, out2, controlled by rop2[15..12], is used only for the ALI – to select a second register as the operand if needed.

**7.5.5 ALU Design**

The arithmetic and logic unit (ALU) is a combinational component that is responsible for carrying out the 4 arithmetic instructions (ADD, SUB, MUL and LSR), and the 3 logic instructions (AND, OR and MVN) in this CPU.

The ALU will have two 16-bit inputs (the operands) as well as a select bus to select which operation is carried out. Internally, a MUX will be used to select and output the result of only one operation at a time. The ALU output will be connected to the accumulator (AC); the 32-bit register whose whole purpose is to hold the result of the ALU.

In order to incorporate the selector bits, two other MUXs known as *operand MUXs,* external to the ALU, will be used to select between the two possible operands (literal/register).

A diagram of a circuit

Description automatically generated

Figure 56 – ALU circuitry. *GPROUT1 and GPROUT2 represent the outputs of two GPR registers.*

**7.5.6 The NZCV flags**

These flags tell the CPU information about the most recent arithmetic instruction, they are defined below:

1. **Negative (N):** this flag is set if the result of the last arithmetic operation was negative; MSB is HIGH (signed arithmetic).
2. **Zero (Z):** this flag is set if the result of the last arithmetic operation was zero.
3. **Carry (C):** this flag is set differently depending on the arithmetic operation being used.
   * ADD instruction: if an overflow occurred.
   * SUB instruction: if a borrow did not occur i.e. operand1 > operand2.
4. **Overflow (V):** similarly, this flag is set differently depending on the arithmetic operation being used.
   * ADD instruction: the addition of two positive numbers, and the result is negative; sign bit (MSB) changes to a 1.
   * SUB instruction: the subtraction of a positive number from a negative number and the result is positive; sign bit changes to a 0.

To implement the semantics of these 4 fundamental CPU flags, a module called *flags\_setter* will be designed. In addition to its obvious input from the accumulator, it would need to know what instruction is currently being executed as only three instructions (CMP, ADD and SUB) set the flags. Furthermore, it would need to know the two operands as well for comparison purposes later.

A drawing of a circuit board

Description automatically generated

Figure 57 – *flags\_setter* module.

**7.5.7 Control Unit Design**

When it is mentioned that the CPU is the “brain” of the computer, what is really meant is the Control Unit (CU). This component acts like traffic lights on an intersection – it controls which *state signals\** are high at which CPU state. It is usually comprised of three parts.

1. **The State Counter**

This is an up-counter that can count to a value equal to the number of CPU states (40 in this case). This is **not** an automatic up-counter (i.e. does **not** count at every clock cycle on its own), instead it is manually controlled by the *Control Signal Logic* module (the third part of the CU). The I/O of this counter and their purpose will now be described.

*Inputs:*

1. **opcode:** connected to the IR.
2. **sel\_bits:** connected to rsel; counter uses this to chooses between multiple paths of a single operation (e.g. mov or ALTmov)
3. **LOAD (control signal):** asserted at the last cycle of the FETCH routine (fetch3). It is at this state that the IR holds the opcode and can be loaded into the *State Counter.* When the opcode is loaded into the *State Counter,* it decides what the counter value would be according to a mapping. The counter can load 16 different opcodes which correspond to the ***first state of each instruction*** (e.g. add1 or mov1).
4. **INCREMENT (control signal):** asserted when the operation’s execution routine state final state. E.g. during fetch1 and fetch2. This is what increments the counter – telling the CPU to move on to the next state.
5. **CLEAR (control signal):** asserted when the operation’s execution routine state final state. E.g. during add2. This serves to return the counter back to the fetch1 state, so the CPU can fetch the next instruction.

*Output:*

1. **q:** the output of the counter fed into the *State Decoder* module (second part of the CU). It is the only output.

A screenshot of a computer program

Description automatically generated

Figure 58 – the 16 different opcodes that may be loaded into the *State Counter.*

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

Figure 59 – counter mapping; each time an opcode is loaded, the counter value is set accordingly.

*A whiteboard with text and a person holding a sign

Description automatically generated with medium confidence*

Figure 60 – *State Counter* module. *6-bits to allow counting to 40.*

1. **The State Decoder**

Connected to the *State Counter,* it takes the counter value and decodes it to one of 40 possible values (for the 40 possible states). Its output is used by the *Control Signal Logic* module which – finally – controls the state signals. Example behaviour of the decoder is shown below.

|  |  |
| --- | --- |
| **6-bit input (from counter)** | **40-bit output (to logic controller)** |
| 0 | 1<<0 |
| 1 | 1<<1 |
| … | … |
| 39 | 1<<39 |

Figure 61 – decoder truth table.

A diagram of a state decoder

Description automatically generated

Figure 62 – *State Decoder* module.

**\* Aside – State Signals (or control signals)**

A state signal is any which can be controlled by the CU. More specifically, it is the output of the *Control Signal Logic* module in the CU. An easy example of state signals are the register load inputs e.g. ARLOAD, ACLOAD or COUNTERLOAD.

*(see next page)*

1. **The Control Signal Logic Controller**

Finally, the module that does the “control” part in the Control Unit. This gigantic combinational lookup table knows which state the CPU is in via the *State Decoder* and sets the correct state signals high, leaving the rest low. **Every control signal along with the state(s) that assert it are shown below.**

A screenshot of a computer program

Description automatically generated

Figure 63 – *State Counter* control signals.

A screenshot of a computer

Description automatically generated

Figure 64 – register loadcontrol signals.

A black and white text

Description automatically generated

Figure 65 – the only increment control signal (increments the program counter).

A white background with black text

Description automatically generated

Figure 66 – memory load control signal. The only two states that load data into memory.

A computer screen shot of a computer code

Description automatically generated

Figure 67 – system bus control signals (discussed in the next section).

A screenshot of a computer

Description automatically generated

Figure 68 – ALU select signals; choice of 7 different arithmetic ALI.

A white card with black text

Description automatically generated

Figure 69 – the *Control Signal Logic* module.

**7.5.8 System Bus Access and Contention**

For register A to send its data to register B, it needs to place its data onto the CPU’s internal system bus (38-bits wide). However, register A is not the only register that may send data elsewhere; many different registers may do that as well. Without controlling which register accesses the system bus in an orderly manner, they will all contend and attempt to place their contents on the bus at the same time – bus contention!

Traditionally, the use of tri-state buffers would be used to manage such contention; ensuring that **only one** register has access to the bus at any given time. This approach will not be taken however due to FPGA synthesis safety concerns that will be discussed in the *Implementation* section.

Alternatively, the use of a MUX (which will be called the *System Bus MUX* or *SB MUX* from now on with its select bus named SYSBUSSEL) will be used to regulate bus access which is semantically equivalent to using tri-state logic.

**7.5.9 Datapath optimisation**

At the beginning of the design phase, it was assumed that every register in the CPU may access the system bus, resulting in a very large *SB MUX.* Instead, upon looking at the execution routines, it was found that only the following **10 registers** need to access the bus at all.

1. PC
2. DR
3. M (memory)
4. TR
5. ROP1
6. ROP2
7. AR
8. AC
9. GPROUT1
10. GPROUT2

Hence, the design of the *SB MUX* can be greatly simplified down to what is shown below.

A diagram of a circuit

Description automatically generated

Figure 70 – *System Bus* used to avoid contention.

**7.5.10 Final CPU Datapath (wiring)**

The final architecture of the CPU is shown below.

A diagram of a circuit

Description automatically generated

Figure 71 – Final CPU architecture.

A diagram of a square with text

Description automatically generated with medium confidence

Figure 72 – The Control Unit.

**7.6 Memory**

Upon reaching this point in the project, the computer would be almost complete, where the only thing missing is memory. But what is meant by memory in this context? *Memory is storage that is external to the CPU.* It can have multiple segments but in this project the term “Memory” is really encapsulation of 3 subtypes of storage:

* **Instruction memory *(read only)*:** serves to hold the instructions that the CPU will fetch and execute.
* **Memory mapped register *(MMRs; mix of read/write and read only):*** allows the CPU to access peripheral control registers via the same address bus (as if it was accessing any other memory).
* **Stack memory *(read/write):*** allows for more permanent storage of data (e.g. variables), as opposed to using its limited number of internal GPRs.

A traditional approach would be to use a bi-directional data bus, which has the ability of writing to memory and reading from memory using a single port. At a low-level this would be using tri-state logic which, as mentioned before, will be avoided for reasons mentioned later on.

Another approach was devised which works around this problem. It involves the use of pre-defined boundary addresses (defining which addresses refer to which of the 3 subtypes of memory) and through the use of an *Address Checker* module, can choose where the data is input and from where it is output. Since a 12-bit address bus was used, the 4096 possible addresses needed to be split amongst the 3 types of memory. So, the following memory map showing the boundary addresses of each was created.

A diagram of a memory stack

Description automatically generated

Figure 73 – The Memory Map.

This memory map allows for fairly large programs to run on the CPU, and interfacing with a plethora of peripherals (~70; as some peripherals would require more than one control register).

**7.6.1 Memory Encapsulation**

Since the single Memory module really has 3 different internal memories, logic had to be designed to control the flow of data such that only one memory location was read from or written to (if permitted). To accomplish this, the following circuit was created.

A diagram of a machine

Description automatically generated

Figure 74 – memory control circuitry. *3 memories can be seen (top to bottom): the instruction memory, the MMR bank, and the stack memory.* ***The MEMLOAD signal is asserted during CPU states str4 and ALTstr4*.**

Again, the use of DEMUX-MUX pairs are used (twice!) here. This arrangement is useful when an array of components are present, and only one needs to be accessed at a time. The purpose of each will now be discussed.

* **The outer pair:** using *Address Checkers,* selects 1 of 3 memories to work with.
* **The inner pair:** using a *Selector* module, chooses 1 out of the 76 MMRs to work with.

The most complex section of this circuitry is the part relating to the MMRs. Thus, the general operation of which will be described below.

*(see next page)*

* **The *Concatenator*:** concatenates the 32-bit data coming from the DR, and the MEMLOAD signal to a single 33-bit bus, where the MEMLOAD signal is carried in bit0 of this bus, and the data itself is held in bits 31..1. This is done to ensure only one MMR is ever loaded at any given time.
* **The *Selector*:** connected to the select busses of the DEMUX-MUX pair, takes the address value, and maps it to one of the MMRs.

**7.6.2 Final Memory Module and Connected MMRs**

So, all in all, the Memory module along with a list of some memory mapped registers to be connected to the CPU is shown below.

A diagram of a diagram

Description automatically generated

Figure 75 – external CPU memory module *(MMR outputs not shown).*

A white screen with black text

Description automatically generated

Figure 76 – list of memory 8 MMRs to be used in this project.

**7.7 LED Panel Designs**

For the casual passer to somewhat appreciate the inner complex beauty of this project, an LED panel was planned. This panel would fully show the contents of all the GPRs and MMRs. Two designs were planned, with only the latter being feasible. Both of which will now be outlined.

**7.7.1 Design 1 – the use of ICs**

This design involved the use of integrated circuits (ICs), external to the FPGA to achieve the desired goal. Example circuitry is illustrated below.

A diagram of a circuit

Description automatically generated

Figure 77 – example circuitry showing 1 LED controller (GPR r0 is shown above). The 8-bit shift register ICs are 74HC595 chips. The clock, reset and serial data (DS) inputs were to come from the FPGA.

The bill of materials (BOM) calculations are shown below:

A screenshot of a computer code

Description automatically generated

Figure 78 – BOM for one LED controller.

A white background with black text

Description automatically generated

Figure 79 – BOM for 23 LED controllers.

A screenshot of a computer code

Description automatically generated

Figure 80 – BOM for “glue circuitry”.

A screenshot of a computer program

Description automatically generated

Figure 81 – final BOM for planned LED panel.

Considering the fact that, for most of these ICs, a DIP/PDIP package was planned to be used, this quickly seemed to be unfeasible. So, another idea was proposed in case.

**7.7.2 Design 2 – the use of WS2812B LEDs**

The WS2812B LED is an *individually addressable LED* that can be cascaded N times, all of which can be controlled separately using a single line. The protocol used to control them is known simply as the *WS2812B Protocol.*

Initially, it was planned to buy these LEDs separately and solder them on a printed circuit board (PCB). However, due to the sheer number of LEDs, and the fact that each LED is a SMD package, this was also deemed unfeasible.

Finally, it was decided that the use of external (pre-assembled) WS2812B LED strips could be used with a custom-built LED panel PCB to connect the FPGA with the strips in a feasible and affordable manner.

**7.8 The Assembler**

The assembler was going to be developed using C++. Every assembly instruction would be its own function. The function itself would be transmitting the relevant bytes over. According to the CPU architecture, the instruction word is 38-bits. The UART itself can only transmit a single byte in one go. So, it was decided that each function would send 6 bytes that would be concatenated in hardware to form the entire 38-bit word. The 6 bytes to be sent (in order of MSB to LSB) are:

1. Opcode
2. Select bits
3. Operand A (HIGH)
4. Operand A (LOW)
5. Operand B (HIGH)
6. Operand B (LOW)

**So, generally, the word structure is:**

**opcode[3..0] s[1..0] a[15..0] b[15..0]**

**8 Implementation, Testing and Problems**

This section is going to cover the actual implementation of the planned design ideas, along with testbench waveforms and any problems that were encountered and how they were overcome. Due to the sheer size of this project, only relevant code snippets are going to be included in this section to provide context for the reader. Otherwise, if the reader is interested in the nuances of the RTL code, they may refer to the author’s GitHub repository which can be found at [*https://github.com/yismailsaadelgedawy/PROJ300*](https://github.com/yismailsaadelgedawy/PROJ300)*.*

**8.1 The VGA Controller**

The HSYNC and VSYNC synchronisation signals were developed first. Again, these are just simple counters.

A screenshot of a computer program

Description automatically generated

Figure 82 – snippet of the *hsync\_cnt* module.

The VSYNC counter was implemented in a similar way. One notable thing however is the difference between when both counters count.

A screenshot of a computer program

Description automatically generated

Figure 83 – HSYNC counter counting **at each positive edge of the pixel clock.**

**A screenshot of a computer program

Description automatically generated**

Figure 84 – VSYNC counter counting **only when the HSYNC counter = 800 (end of scanline).**

Next, the *RGB Controller* module was designed, which was responsible for outputting colour to the display.

A computer screen shot of a code

Description automatically generated

Figure 84 – assigning 4-bits/colour from a 12-bit colour input.

The key to this module is to have the output colour data off (0x000) when the scanline is not in the active region. A simple XNOR of the rgb\_en outputs from the HSYNC and VSYNC counters achieve just that.

A screenshot of a computer program

Description automatically generated

Figure 85 – VGA Controller wiring.

For now, the colour data would come from a single 4-pixel sprite (square).

A computer code with numbers and symbols

Description automatically generated

Figure 86 –sprite ROM initialisation holding the colour data.

A screenshot of a computer program

Description automatically generated

Figure 87 – sprite output logic; this is where the coordinates are defined. The internal offset registers receive their inputs from the *Pixel Offset Controller* module.

The sprite module has a 44-bit output, pixel\_pos, corresponding to the 11-bit coordinate data of the 4-pixels used for edge detection. This data was then fed to the offset controller.

In order to facilitate the movement of the sprite, the *Pixel Offset Controller* was built next.

A screenshot of a computer code

Description automatically generatedA screenshot of a computer program

Description automatically generated

Figure 88 – the 4-bit input could come from anywhere. For testing purposes, the 4 push-buttons on the FPGA development board was used. *Note the edge detection used!*

**Problems:**

* Initially, 10-bits were used for the coordinate data (max. value of 1023) to accommodate for the maximum column coordinate of 800. However, as signed logic had to be used to accommodate for negative coordinate offsets, the width had to be increased to11-bits to account for the sign bit.
* The pixel clock is 25MHz, the *Pixel Offset Controller* clock had to be slowed down otherwise the sprite movement would be too fast to be usable. This meant clock domain synchronisation (CDC) had to be used to avoid metastability issues.

A diagram of a computer

Description automatically generated

Figure 89 – CDC chains before sending the data to the sprite ROM(s).

A green lines on a black background

Description automatically generated

Figure 90 – *hsync\_cnt* testbench results.

A screen shot of a computer

Description automatically generated

Figure 91 – *vsync\_cnt* testbench results. *Note it is only counting when the column input is 800.*

The following testbench showcases the sprite rendering with row and column offsets of -50 and +50 respectively. The original coordinates of the sprite were:

A number of numbers on a white background

Description automatically generated

Figure 92 – initial sprite coordinates.

So, the expected coordinates of the sprite with offsets would be

A number of numbers on a white background

Description automatically generated

Figure 93 – offset sprite coordinates.

**

Figure 94 – correct colour data output at the correct row and column coordinates.

A red and green squares

Description automatically generated

Figure 95 – what the sprite should look like.

A black screen with a small red and green dot

Description automatically generated

Figure 96 – real life test; what the sprite looks like!

A computer screen with a blue background

Description automatically generated

Figure 96 – initial sprite position.

A screen with a white background

Description automatically generated with medium confidence

Figure 97 – movement of sprite via user input.

**8.2 The PS/2 Controller (Keyboard)**

First, the *ps2\_controller* module was built, which decoded the incoming data frame and stored the relevant serial data in a SPSR.

A screenshot of a computer program

Description automatically generated

Figure 98 – snippet of controller module.

Next, a *bsy\_detector* module was built, which detects the busy flag going low indicating eot. Then, the data output of the SPSR would be latched by this module and can be used.

A computer code with colorful text

Description automatically generated

Figure 99 – *bsy\_detector* module code.

**Problems:**

* Whilst these two modules alone would work, it was soon realised that key release detection was not incorporated. Meaning that once a key is pressed, its make code would appear on the output and would remain latched even if the key was released. To remedy this, a *break\_code\_detector* module was developed.
* When creating this module initially, a reset signal would be pulsed to the *bsy\_detector* module to reset the latch. This did not work. This was because the break code sent upon key release was not just 0xE0, but it was also followed by another byte – the make code of the key that was pressed. To fix this, the reset pulse had to be longer than a data frame once the 0xE0 byte was detected.

A screenshot of a computer program

Description automatically generated

Figure 100 – *break\_code\_detector* module code.

A diagram of a computer

Description automatically generated

Figure 101 –PS/2 Keyboard Controller wiring. *Note the OR of the two reset signals.*

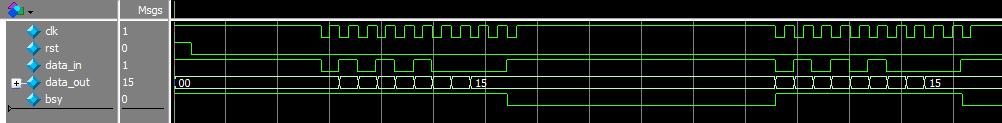
**

Figure 102 – *ps2\_controller* module testbench results. This involved simulating the press of key ‘Q’ (make code 0x15) twice.

A hand holding a keyboard and a circuit board

Description automatically generated

Figure 103 –real-life testing; showcasing the letter ‘A’ (make code 0x1C) being pressed.

**8.3 The Audio Controller**

First, the *key\_detector* module was built. As mentioned before, this was just a lookup table which outputs the select value for the MUX that chooses which musical frequency to output. Initially, this Audio Controller was hard-wired to the output of the PS/2 controller for testing purposes.

A screenshot of a computer program

Description automatically generated

Figure 104 –snippet of the sharp keys lookup table.

A screenshot of a computer code

Description automatically generated

Figure 105 –snippet of the normal keys’ lookup table.

A white background with black text

Description automatically generated

Figure 106 –if no key is pressed, output select value 0 (corresponds to OFF).

**Problem:** when looking at simple music to play after the implementation of the Audio Controller, it was realised that many times, a lower octave was required – an oversight in the original design. So, to overcome this issue, the RShift key (make code 0x59) would be used to toggle between octave IV and the lower octave III. This obviously meant that more frequencies had to be stored on the FPGA fabric but that was not an issue considering the *­­fine\_clk\_divN\** module used to generate these frequencies barely used any resources.

With that in mind, the *special\_key\_detector* was built next.

A screenshot of a computer program

Description automatically generated

Figure 107 –toggling of output flags when special keys were pressed. *This idea was inspired by Sticky Keys in Microsoft’s Windows OS.*

\* **Aside – the Fine Clock Divider Module**

The well-known method of creating a clock divider is through the use of an N-bit up-counter with an input frequency . The output frequency signal, , would be hard-wired to the MSB of the counter, **bit N-1**. Leading to the following definition:

To understand why this may be a problem in some cases consider the example below.

Assuming a 50 MHz input clock, and with N=16:

And with N=17:

So, a natural question is: what if a frequency in between N=16 and N=17 is required? For example, 500 Hz.

The solution to this problem is to create a more accurate clock divider module which will be referred to as the *fine clock divider.* To create this, a few derivations need to be made.

Assuming a sufficiently wide counter that may count to a value N, with an input frequency, . It is natural to assume, that if the counter is fed a frequency of 1 Hz, and can count to a value of 100, it will take 100 seconds to reach that value. I.e.

Furthermore, if the input frequency was upped to 2 Hz:

So, with this basic intuition, a more a general equation may be derived:

This means that it is possible to have an internal signal that toggles its state once the counter reaches that defined value, N. However, the counter must reach that value twice to complete a full cycle *(toggling from low 🡪 high, then high 🡪 low).* So,

With that, a **parameterisable** *fine clock divider* module can be created, where the user can input their desired output frequency, , and get just that. Thus, given a desired output frequency, the value that the counter would need to count to using a 50 MHz input clock is:

**A screenshot of a computer program

Description automatically generated**

Figure 108 – *fine\_clk\_divN* module code.

A screenshot of a computer

Description automatically generated

Figure 109 – *fine\_clk\_divN* testbench waveform with the previously unattainable frequency of f=500 Hz (T=2 ms; see cursor time delta).

*To evaluate the accuracy of this clock divider as opposed to the simpler version, a graph of is shown below.*

A graph with a red line

Description automatically generated

Figure 110 –graph of the relationship between and @ (normalised to 1 MHz).

For low values of N (higher desired frequencies), the clock divider is relatively inaccurate, but as N increases (lower desired frequencies), so does its precision.

Its precision is best illustrated by an examples (see next page).

**Low N example:**

**Large N example:**

With the *fine\_clk\_divN* module now built, an array of them can be created, multiplexed, and chosen according to the key input from the keyboard. See the example below.

A screenshot of a computer program

Description automatically generated

Figure 111 –array of musical frequencies in octave IV (according to Figure 9).

An array representing octave III was constructed in a similar manner, and all frequencies (found in the *music\_frequencies* module below) where then multiplexed for later use. The complete Audio Controller circuitry is shown below.

A computer screen shot of a diagram

Description automatically generated

Figure 112 – Audio Controller module wiring.

For testing purposes, the Audio Controller received its inputs from the PS/2 keyboard via hard-wiring.

A diagram of a graph

Description automatically generated with medium confidence

Figure 113 – Audio Controller module wiring.

A close-up of a circuit board

Description automatically generated

Figure 114 – PWM buzzer playing note A when key ‘A’ (make code 0x1C) is pressed.

**8.4 The UART**

**8.4.1 Protocol Implementation**

To start, the UART’s sampling clock had to be created. As mentioned in the design section, this clock would only start once it senses the start bit and ends when it senses the stop bit. This module is a modified version of the *fine\_clk\_divN* module discussed earlier, so overall the semantics are similar.

A screenshot of a computer program

Description automatically generated

Figure 115 – snippet of the ­*uart\_clk\_divN* module. The clock only starts when transmission\_state is set. sense is connected to the data line, and bsy is connected to the *uart\_controller* module itself.

The rest of the implementation is nearly identical to the *PS2\_controller* module, so there is no need for an in-depth discussion.

A computer diagram with many lines and text

Description automatically generated with medium confidence

Figure 116 – the UART wired up. Parameterisable so support for any baud rate is included! *N.B. This only implements the receiver.*

The following testbench shows the ASCII character ‘J’ being sent to the UART module twice. Note the bsy line pulsing low – indicating eot.



Figure 117 – UART testbench waveform. Data is not ready to be ready until the internal SPSR is done shifting its contents.

**Problems:**

* A fairly complex instruction buffer needed to be constructed that can concatenate 6-byte words into the desired 38-bit instruction.
* The CPU instruction word is intended to be 38-bits. There are only 6 seven segment displays on the FPGA development board which can display 3 bytes at a time. There were two methods of tackling this issue. The first would be to purchase more seven segment displays so that all 6 bytes can be shown at once. The other option would be to use a debug button on the board to switch between viewing the top 3 bytes of the word, and the bottom 3 bytes whilst the button was held down. The latter was chosen.
* Another debug button would be needed to increment the address of the buffer to view its contents. The synchronous buffer would be running at 50 MHz so switch bounce would be an issue; hardware debouncing would be required.

**8.4.2 The Instruction Buffer**

Before tackling any of the above issues, a choice had to be made on how to implement address control, i.e. how would the buffer know to write to the next memory location after the data was received? Two methods will now be outlined.

1. **Special Byte Detection**

This method was initially implemented where, a special character (for example ASCII ‘$’) would be sent to tell the buffer to increment its internal address counter. Whilst this did work, this method effectively halved the speed at which data was being sent due to every byte of information requiring an additional “incrementor byte”. The speed at which data was being sent is crucial due to a *single instruction requiring 6 bytes* to be transmitted, and a program would potentially require a lot of instructions. Another method was used instead.

1. **Using The Eot Flag**

Admittedly, this solution was pretty obvious from the start and the prior method was very over-engineered. The *bsy\_detector* module was modified to output an active high eot flag as shown below.

A screenshot of a computer code

Description automatically generated

Figure 118 – non-latching eot flag.

The buffer can now use the eot pulse to increment its internal address counter. Other than its simplicity, this method **doubles the speed of transmission** of data due to the removal of the “incrementor byte”.

Next, the buffer itself would be mad of two parts: a smaller intermediary RAM, and the bigger instruction RAM. Recall that the instruction would be sent in 6-byte chunks in the following order:

1. Opcode
2. Select bits
3. Operand A (HIGH)
4. Operand A (LOW)
5. Operand B (HIGH)
6. Operand B (LOW)
7. **The Intermediary RAM**

This buffer would be precisely 6\*1-byte memory locations deep, one address for each byte above. It would also have 6\*1-byte outputs hard-wired to these memory locations. These outputs would then be the input to the actual instruction RAM.

A computer code with black text

Description automatically generated with medium confidence

Figure 119 – module IO ports.

A screenshot of a computer program

Description automatically generated

Figure 120 – RAM creation and output assignments.

A very important line is the rdy output. This line connects to the instruction RAM to instruct it to concatenate the 6-bytes together into one 38-bit word.

A screenshot of a computer program

Description automatically generated

Figure 121 – write semantics of the intermediary RAM. *Note the rdy flag pulsing once the 6-bytes were received.*

The actual instruction RAM that ultimately holds all the instructions the CPU will fetch from is quite simple.

A white background with text

Description automatically generated

Figure 122 – storing the concatenated data into the instruction RAM.

To allow for switching between the top and bottom 3 bytes:

A screenshot of a computer program

Description automatically generated

Figure 123 – choosing which 3 bytes to output.

To solve switch bouncing, a hardware debouncer was created. As soon as its input goes high, this module mimics the input after a defined time delay. There are 3 states the *debouncer* module can be in: 1) idle, 2) pressed and 3) released.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer code

Description automatically generated

Figure 124 – *debouncer* implementation; 100ms was found to be more than sufficient.

**Problem:** during testing, the displays would show zero whenever the increment debug button was pressed. It was later found that, upon button press, it would not display the contents of the next address, instead it would jump to a very large empty memory location. This module would be running at a very fast clock and would register hundreds of button inputs in the duration it was pressed in. To combat this, a “guard” had to be used in the *instructionRAM* module to ensure it increments the address only once.

A computer code with text

Description automatically generated

Figure 125 – making use of Verilog’s non-blocking assignment feature!

*N.B. despite the instructionRAM module operating at 9600 Hz as well, it is* ***NOT*** *in the same clock domain as the UART (different phases are probable). So, the use of CDC was needed to avoid potential metastability.*

A computer screen shot of a computer program

Description automatically generated

Figure 126 – *instructionBuffer* module fully wired.

A diagram of a computer circuit

Description automatically generated

Figure 127 – *debouncer* modules in use for the debug buttons.

A screenshot of a computer program

Description automatically generated

Figure 128 – UART and buffer connected. *(CPU address line was grounded for testing; CPU was not built at this stage).*

A screenshot of a computer program

Description automatically generated

Figure 129 – *instructionRAM* testbench with a maximum of 4 instructions.

There is a lot going on in the above waveforms so it will be explained.

* When load goes high, the 6 bytes are concatenated, and the instruction is stored in the first memory location since the address input is 0. Note how ram[37..34], which holds the 4 opcode bits holds 0001, and ram[33..32] holds the 2 select bits 01.
* When debug btnB is low (not pressed), the top 3 bytes are shown
  + The opcode: 0x01
  + The select bits: 0x01
  + Operand A (HIGH): 0x00
* When debug btnB is high (pressed), the bottom 3 bytes are shown instead
  + Operand A (LOW): 0x02
  + Operand B (HIGH): 0x00
  + Operand B (LOW): 0x03
* **So overall the instruction contains**
  + The opcode: 4’d1
  + The select bits: 2’b01
  + Operand A: 16’d2
  + Operand B: 16’d3

A screenshot of a computer

Description automatically generated

Figure 130 – *debouncer* testbench. Note how the output q goes high 100ms after the input d goes high.

**Problem – Memory Synthesis and BRAM inference**

The FPGA fabric has many internal components that can be wired together upon synthesis of RTL code. This includes Lookup Tables (LUTs), embedded multipliers and so on. One component of particular relevance here is Block RAM (BRAM). BRAM is an embedded memory block within the FPGA. Using BRAM means that no Adaptive Logic Modules (ALMs\*) will be utilised, saving logic resources for everything else. However, to infer BRAM from RTL code, it is recommended that an Intellectual Property (IP) block is used. Otherwise, extremely specific RTL syntax must be followed to allow the synthesis tools to infer BRAM for the programmer.

The problem with inferring BRAM using RTL code, is that the BRAM module must not contain any other logic otherwise the synthesis tools cannot infer the memory block and will use ALMs instead. This leads to **huge** amounts of logic utilisation and **very** long compilation times.

As both RAM modules’ RTL code in this section were not purely RAM synthesis, but included other logic as well in the same code file, BRAM was not inferred and just 100 memory locations in the *instructionBuffer* (max 100 instructions) lead to a **massive 11% ALM utilisation** and took **several minutes** to compile.

A screenshot of a computer

Description automatically generated

Figure 131 – ALMs used instead of BRAM.

To fix this, the logic could be externalised to a syntax-abiding BRAM module. An example is shown below.

A diagram of a computer program

Description automatically generated

Figure 132 – example of how logic controlling the BRAM may be externalised.

Verilog RTL syntax example:

A screenshot of a computer

Description automatically generated

Figure 133 – Verilog “ramstyle” attribute (Intel Quartus only) [18].

Instead, IP blocks may be initialised and used:

A screenshot of a computer program

Description automatically generated

Figure 134 – Intel Quartus IP Catalog.

It is important to note that, if the maximum number of available BRAM bits are exceeded, ALMs will then be used. Each FPGA has a limited number of BRAM bits that may be used. The DE0-CV board uses the Cyclone V 5CEB**A4**F23C7N FPGA which has the following resources:

A white sheet with yellow text

Description automatically generated

Figure 135 – Cyclone V resource availability [19].

Intel Quartus “ramstyle” Definitions [20]:

* **M9K** – memory block containing **9kb** of memory.
* **M10K** – memory block containing **10kb** of memory.
* **M20K** – memory block containing **20kb** of memory.
* **M144K** – memory block containing **144kb** of memory.

*\* ALMs are also known as Logic Elements (LE).*