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

1 Abstract

This project involves designing and implementing a full computer whose heart is a fully programmable 32-bit CPU that can 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 two 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 including a 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

AC – Accumulator

AL – Arithmetic and Logic (instruction)

ALM – Arithmetic Logic Module

ALT – Alternative (instruction)

ALU – Arithmetic Logic Unit

AR – Address Register

bps – Bits Per Second

BRAM – Block RAM

CMOS – Complementary Metal-Oxide-Semiconductor

CPU – Central Processing Unit

CU – Control Unit

DAC – Digital-to-Analog Converter

DEMUX – Demultiplexer

DR – Data Register

E/H field – Electric/Magnetic field

eot – End of Transmission

FPGA – Field-Programmable Gate Array

GI – General Instruction

GPR – General Purpose Register

HSYNC – Horizontal Synchronization

I/O (IO) – Input/Output

IC – Integrated Circuit

IP – Intellectual Property

IR – Instruction Register

ISA – Instruction Set Architecture

LE – Logic Element

LSB – Least Significant Bit

LUT – Look-Up Table

M – Memory

MCU – Microcontroller Unit

MMR – Memory-Mapped Register

MSB – Most Significant Bit

MUX – Multiplexer

NZCV – Negative, Zero, Carry, Overflow Flags (in ARM architecture)

PC – Program Counter

PSSR – Parallel-to-Serial Shift Register

PSU – Power Supply Unit

PWM – Pulse Width Modulation

RAM – Random Access Memory

RISC – Reduced Instruction Set Computer

ROM – Read-Only Memory

RS232 – Recommended Standard 232

RTL – Register Transfer Level

Rx – Receive

sot – Start of Transmission

SPSR – Serial-to-Parallel Shift Register

TTL – Transistor-Transistor Logic

Tx – Transmit

UART – Universal Asynchronous Receiver/Transmitter

VGA – Video Graphics Array

VSYNC – Vertical Synchronization

4 Introduction

Embedded systems and electronics tend to always be seen as this “magical black 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.

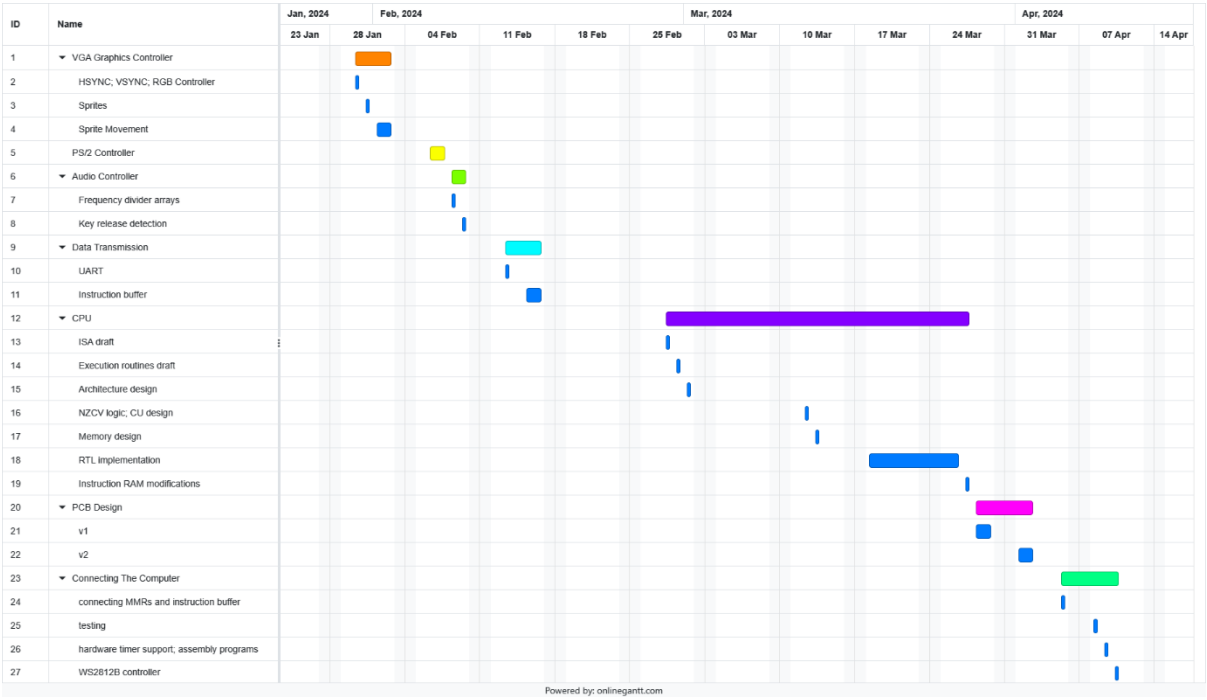
In order to design and build complicated systems, simpler versions must be constructed or at least fully understood, which is precisely the approach the author took on this project. A barebones CPU, with merely 4 instructions and a few basic registers was first planned, tested, and synthesized on an 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 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.* To realise this, one working week was spent implementing each peripheral (1 week for each of the: 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.

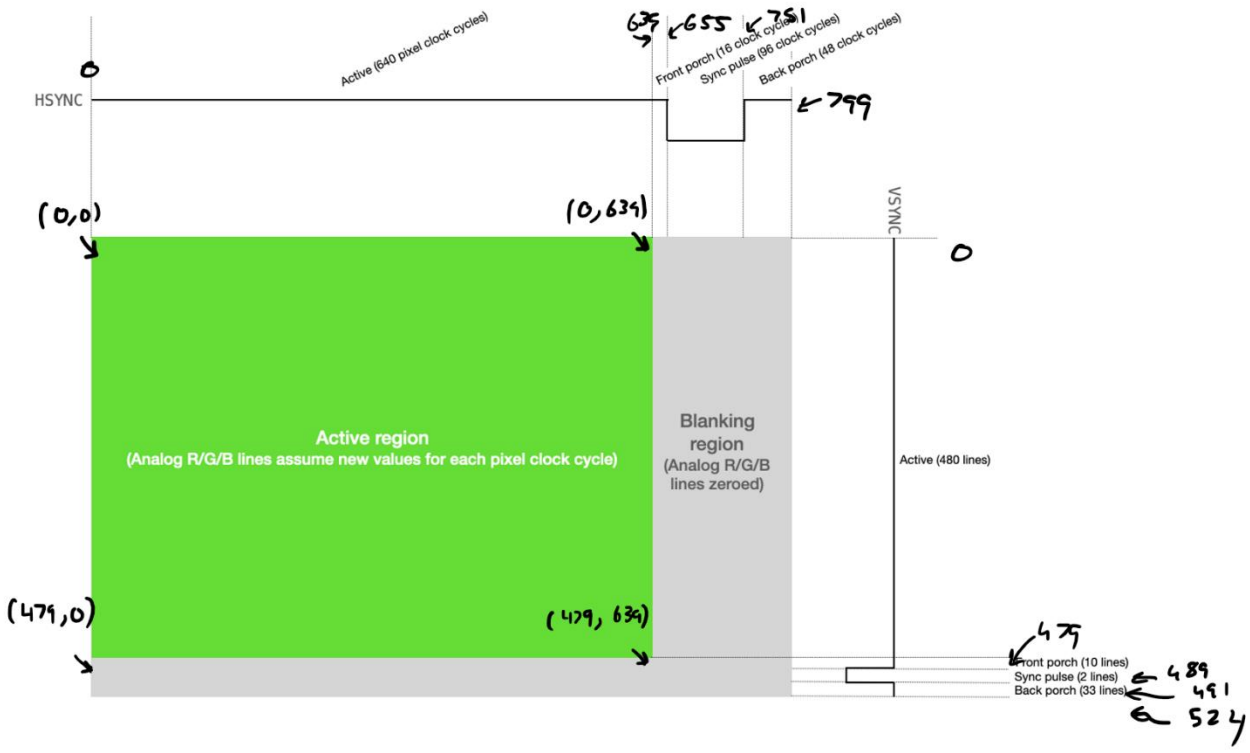


Figure 1 – VGA protocol diagram for a 640x480 resolution [1] with the author's own annotations where the numbers in brackets represent *(rows, columns).*

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.

VGA Timings

The following table lists timing values for several popular resolutions.

Format	Pixel Clock (MHz)	Horizontal (in Pixels)				Vertical (in Lines)			
		Active Video	Front Porch	Sync Pulse	Back Porch	Active Video	Front Porch	Sync Pulse	Back Porch
640x480, 60Hz	25.175	640	16	96	48	480	11	2	31
640x480, 72Hz	31.500	640	24	40	128	480	9	3	28
640x480, 75Hz	31.500	640	16	96	48	480	11	2	32
640x480, 85Hz	36.000	640	32	48	112	480	1	3	25
800x600, 56Hz	38.100	800	32	128	128	600	1	4	14
800x600, 60Hz	40.000	800	40	128	88	600	1	4	23
800x600, 72Hz	50.000	800	56	120	64	600	37	6	23
800x600, 75Hz	49.500	800	16	80	160	600	1	2	21
800x600, 85Hz	56.250	800	32	64	152	600	1	3	27
1024x768, 60Hz	65.000	1024	24	136	160	768	3	6	29
1024x768, 70Hz	75.000	1024	24	136	144	768	3	6	29
1024x768, 75Hz	78.750	1024	16	96	176	768	1	3	28
1024x768, 85Hz	94.500	1024	48	96	208	768	1	3	36

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.

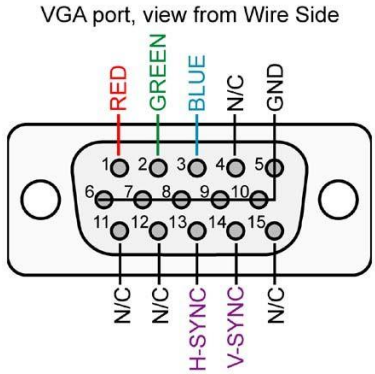


Figure 4 – DB-15 pinout [5].

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.

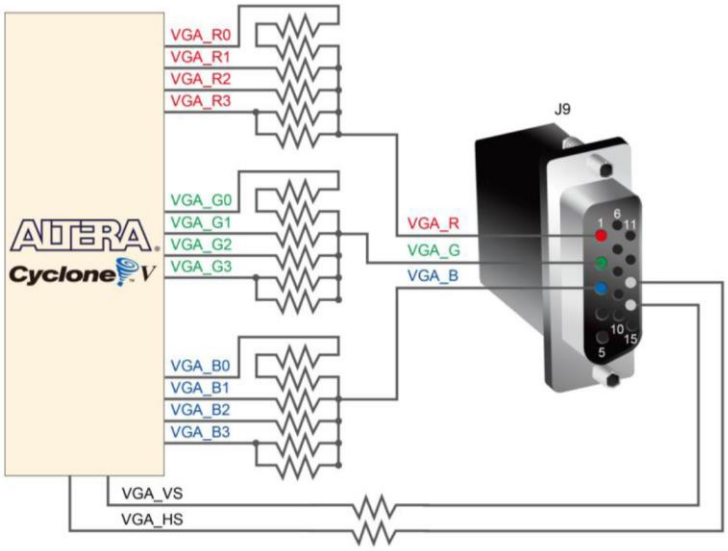


Figure 5 – DB-15 pinout on the DE0-CV board, taken from its user manual [8].

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.

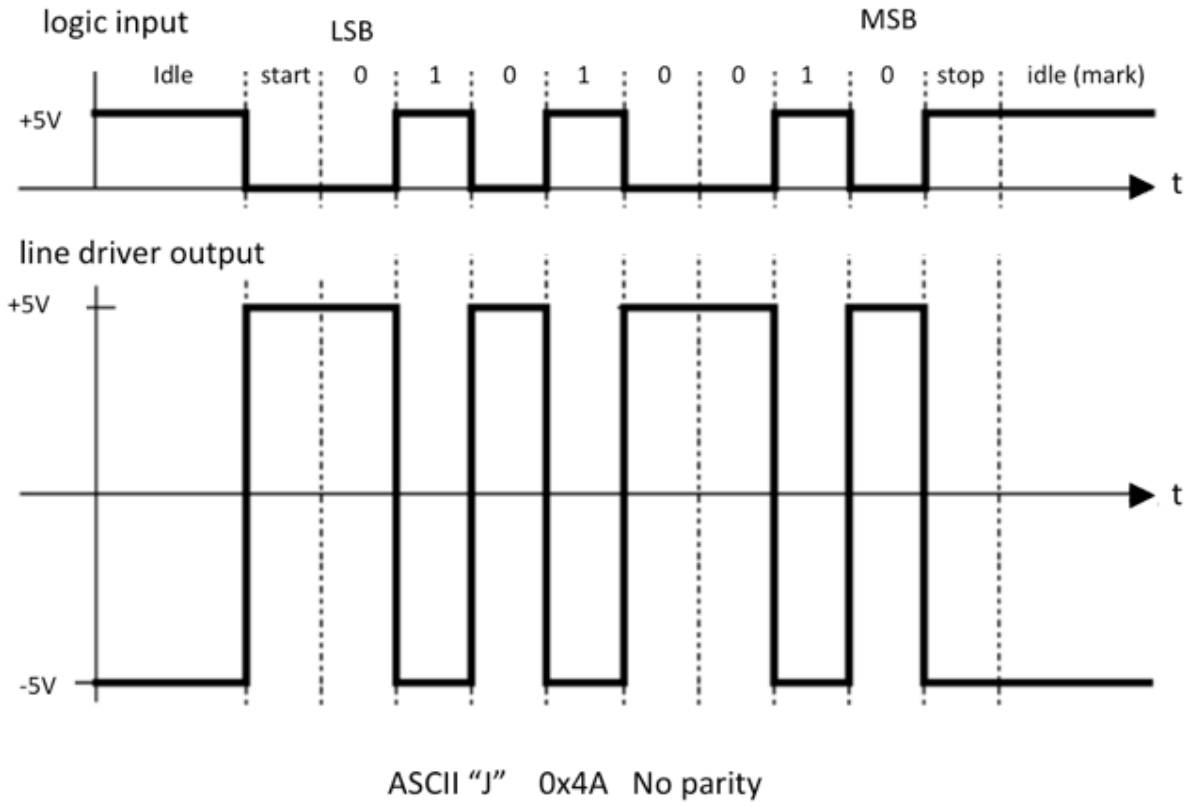


Figure 6 – RS232 frame structure. Top: logic input (and output if no line driver is used). Bottom: line driver output [9].

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 it was, 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.*

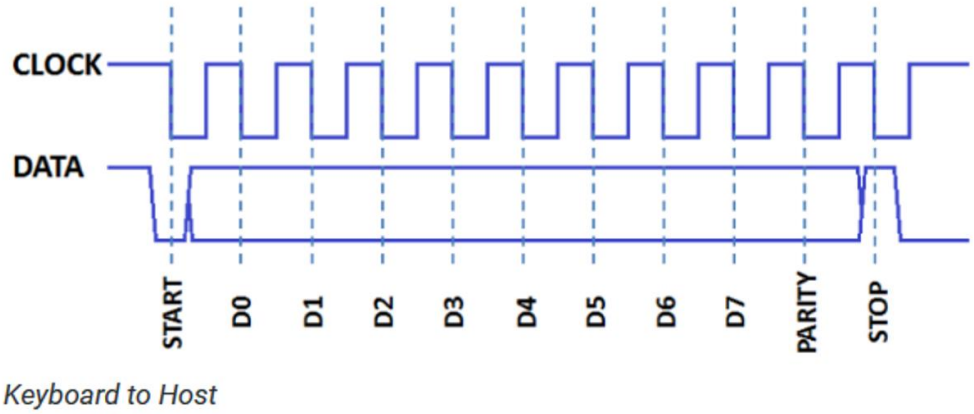


Figure 7 – PS/2 keyboard frame structure [10].

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.

KEY	MAKE CODE	KEY	MAKE CODE	KEY	MAKE CODE	KEY	MAKE CODE	KEY	MAKE CODE
A	1C	N	31	0	45	`	0E	F1	05
B	32	O	44	1	16	-	4E	F2	06
C	21	P	4D	2	1E	=	55	F3	04
D	23	Q	15	3	26	\	5D	F4	0C
E	24	R	2D	4	25	KP /	E0, 4A	F5	03
F	2B	S	1B	5	2E	KP *	7C	F6	0B
G	34	T	2C	6	36	KP -	7B	F7	83
H	33	U	3C	7	3D	KP +	79	F8	0A
I	43	V	2A	8	3E	[	54	F9	01
J	3B	W	1D	9	46	]	5B	F10	09
K	42	X	22	.	49			F11	78
L	4B	Y	35	,	41			F12	07
M	3A	Z	1A	;	4C				

KEY	MAKE CODE	KEY	MAKE CODE	KEY	MAKE CODE
Backspace	66	Left SHIFT	12	Up Arrow	E0, 75
SPACE	29	Right SHIFT	59	Left Arrow	E0, 6B
TAB	0D	Left CTRL	14	Down Arrow	E0, 72
Caps Lock	58	Right CTRL	E0, 14	Right Arrow	E0, 74
ENTER	5A	Left ALT	11		
ESC	76	Right ALT	E0, 11		
DEL	E0, 71	INSERT	E0, 70		

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

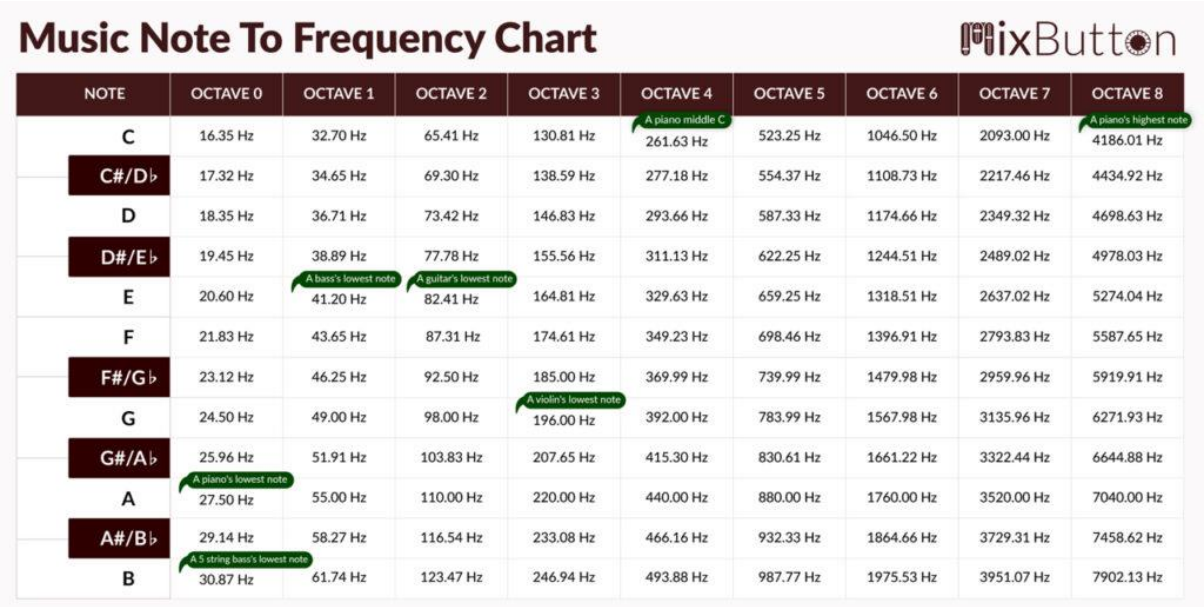
In this project, the keyboard is the “host”, and the FPGA is the “device”. So far, only the data sent from host to device has been covered. The protocol of sending special bytes (commands) from device to host was not implemented and will not be explained but a table of commands are listed below.

COMMAND	COMMAND CODE (HEX)	REMARKS
Reset	FF	Resets the keyboard.
Resend	FE	Resend the last sent byte.
Set Default	F6	Load default keyboard settings.
Disable	F5	Stop the keyboard from sending scan codes.
Enable	F4	Enable the keyboard after being disabled.
Set Typematic Rate	F3	Following this command another byte is send to adjust the typematic rate and delay.
Echo	EE	Keyboard responds with 0xEE.
Set Reset LED’s	ED	Following this command another byte is send to enable/disable the LED’s. Bit 0 = Scroll Lock, Bit 1 = Num Lock, Bit 2 = Caps Lock.

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 in the project, the specific frequencies of these notes must be known.



Music Note To Frequency Chart

MixButton

NOTE	OCTAVE 0	OCTAVE 1	OCTAVE 2	OCTAVE 3	OCTAVE 4	OCTAVE 5	OCTAVE 6	OCTAVE 7	OCTAVE 8
C	16.35 Hz	32.70 Hz	65.41 Hz	130.81 Hz	A piano middle C 261.63 Hz	523.25 Hz	1046.50 Hz	2093.00 Hz	A piano's highest note 4186.01 Hz
C#/D♭	17.32 Hz	34.65 Hz	69.30 Hz	138.59 Hz	277.18 Hz	554.37 Hz	1108.73 Hz	2217.46 Hz	4434.92 Hz
D	18.35 Hz	36.71 Hz	73.42 Hz	146.83 Hz	293.66 Hz	587.33 Hz	1174.66 Hz	2349.32 Hz	4698.63 Hz
D#/E♭	19.45 Hz	38.89 Hz	77.78 Hz	155.56 Hz	311.13 Hz	622.25 Hz	1244.51 Hz	2489.02 Hz	4978.03 Hz
E	20.60 Hz	A bass's lowest note 41.20 Hz	A guitar's lowest note 82.41 Hz	164.81 Hz	329.63 Hz	659.25 Hz	1318.51 Hz	2637.02 Hz	5274.04 Hz
F	21.83 Hz	43.65 Hz	87.31 Hz	174.61 Hz	349.23 Hz	698.46 Hz	1396.91 Hz	2793.83 Hz	5587.65 Hz
F#/G♭	23.12 Hz	46.25 Hz	92.50 Hz	185.00 Hz	369.99 Hz	739.99 Hz	1479.98 Hz	2959.96 Hz	5919.91 Hz
G	24.50 Hz	49.00 Hz	98.00 Hz	A violin's lowest note 196.00 Hz	392.00 Hz	783.99 Hz	1567.98 Hz	3135.96 Hz	6271.93 Hz
G#/A♭	25.96 Hz	51.91 Hz	103.83 Hz	207.65 Hz	415.30 Hz	830.61 Hz	1661.22 Hz	3322.44 Hz	6644.88 Hz
A	A piano's lowest note 27.50 Hz	55.00 Hz	110.00 Hz	220.00 Hz	440.00 Hz	880.00 Hz	1760.00 Hz	3520.00 Hz	7040.00 Hz
A#/B♭	29.14 Hz	58.27 Hz	116.54 Hz	233.08 Hz	466.16 Hz	932.33 Hz	1864.66 Hz	3729.31 Hz	7458.62 Hz
B	A 5 string bass's lowest note 30.87 Hz	61.74 Hz	123.47 Hz	246.94 Hz	493.88 Hz	987.77 Hz	1975.53 Hz	3951.07 Hz	7902.13 Hz

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 *http://www.ece.uprm.edu/~jnavarro/sample.pdf*

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:

4. **Address Register (AR):** stores the address of the current instruction to be fetched from memory.
5. **Program Counter (PC):** stores the address of the next instruction to be fetched from memory.
6. **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.
7. **Accumulator (AC):** stores the result of any arithmetic and logic operation on operands (output from the ALU).
8. **Instruction Register (IR):** stores the opcode portion of the instruction word.

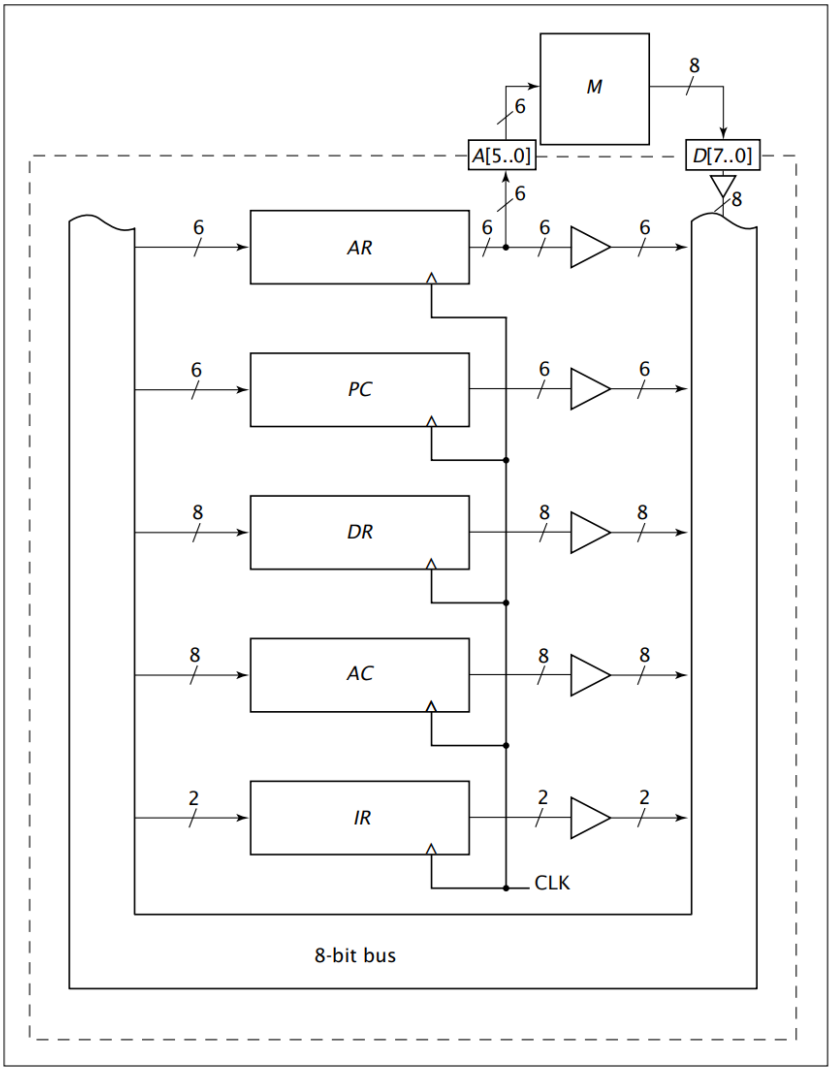


Figure 10 – example architecture of a simple CPU showing the fundamental registers.

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/time unit).
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 and when) according to which state the CPU is in. It generally consists of 3 parts:
    a. ***State Counter:*** *holds current CPU state as a numerical value.*
    b. ***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 set).*
    c. ***Combinational Logic Controller:*** *takes its input from the decoder and generates the correct state signals accordingly as well as controlling 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.

```
FETCH1 : AR <- PC
FETCH2 : DR <- M; PC <- PC + 1
FETCH3 : IR <- DR[7..6]; AR <- DR[5..0]
```

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 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 is outlined below.

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 was to use a toggle flip-flop, whose output toggles at a frequency that is exactly half the input frequency used as its clock.

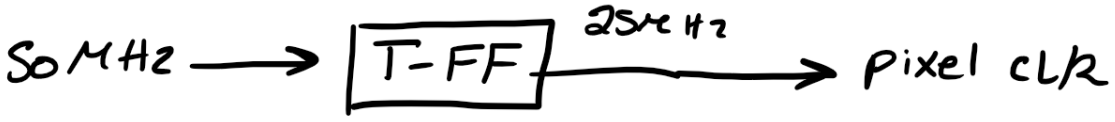


Figure 12 – T flip-flop block diagram. *Note: the input, T, is to be held constantly high (at Vcc).*

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.

a) **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.
b) **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
```

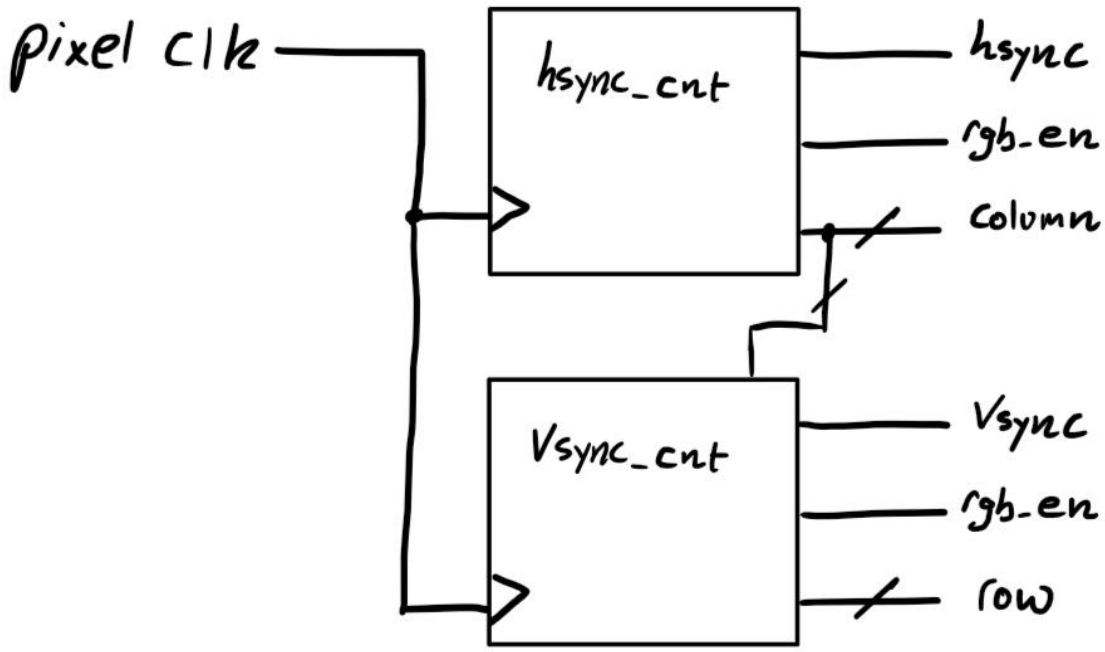


Figure 12 – block diagram of synchronisation signals. The “row” and “column” are the real-time coordinate of the scanline. The “rgb_en” outputs will be connected to the RGB controller (see the next section).

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.

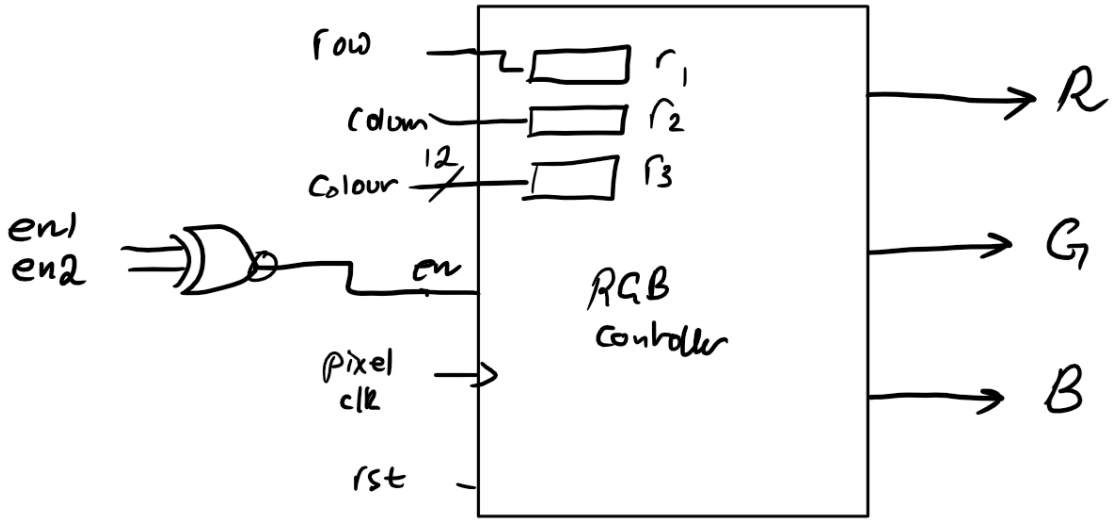


Figure 13 – block diagram of the RGB controller.

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 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.

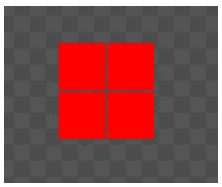


Figure 14 – 4-pixel square sprite. *This sprite was made by* http://www.piskelapp.com

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 a 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.

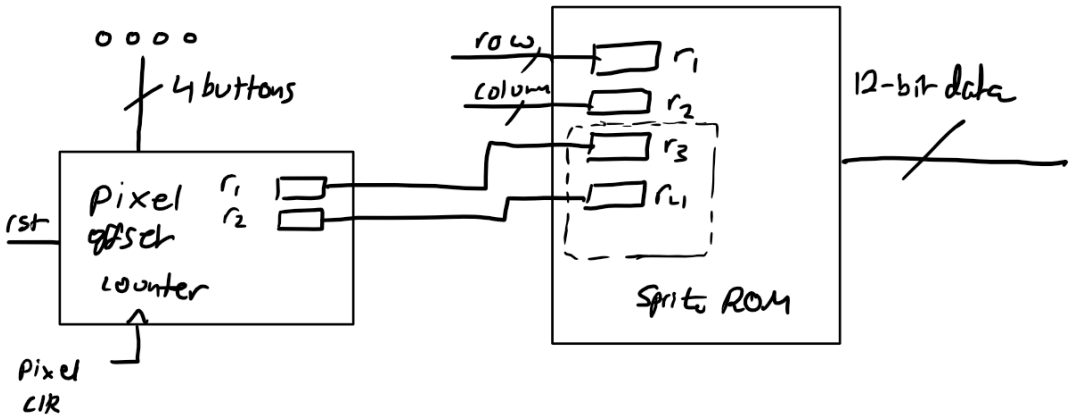


Figure 15 – sketch of proposed idea.

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 pixel** 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.

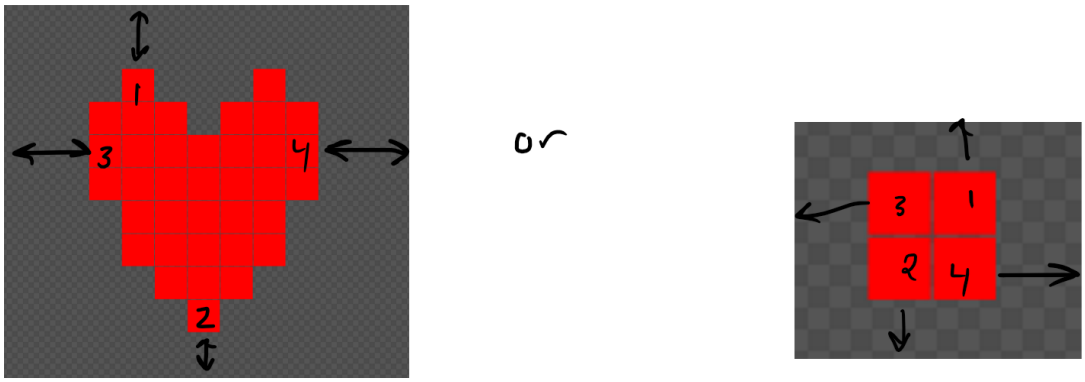


Figure 16 – example sprites with edge detection of 4 “vertex pixels”. *These sprites were made by* http://www.piskelapp.com

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 and 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.

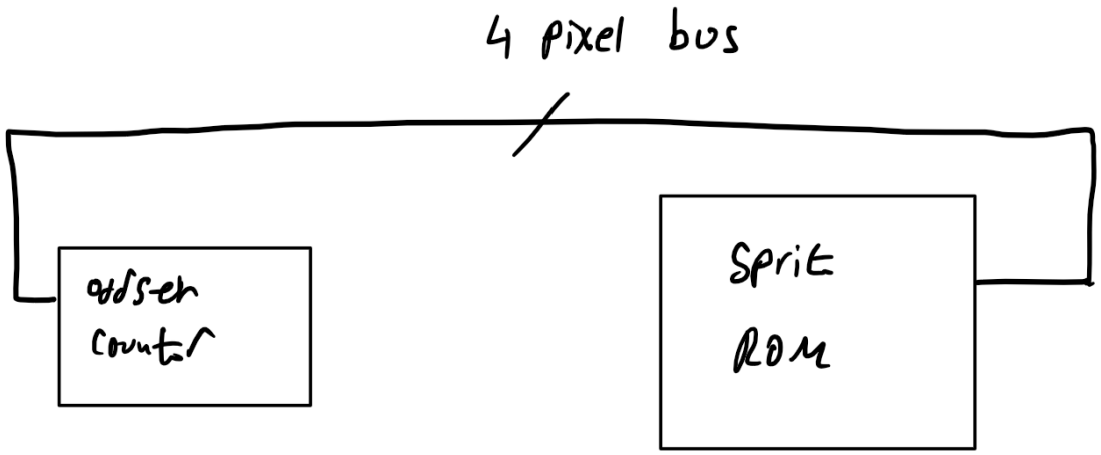


Figure 17 – sketch showing how the 4-pixel coordinate data would be wired.

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++.
    a. count = 0; idle.

b. count = 1; start bit.
c. $2 \leq$ count ≤ 9; data frame – this is the only time the data is placed inside a SPSR for storage.
d. count = 10; odd parity bit.
e. count = 11; stop bit – use this as a reset back to count = 0 to signify eot (end of transmission).
f. An internal SPSR (serial-parallel shift register) will be required.
g. A busy flag will be used as follows:
h. busy = 0 when count = 0.
i. busy = 1 for entire frame.
j. then busy = 0, and data may be read off the SPSR.

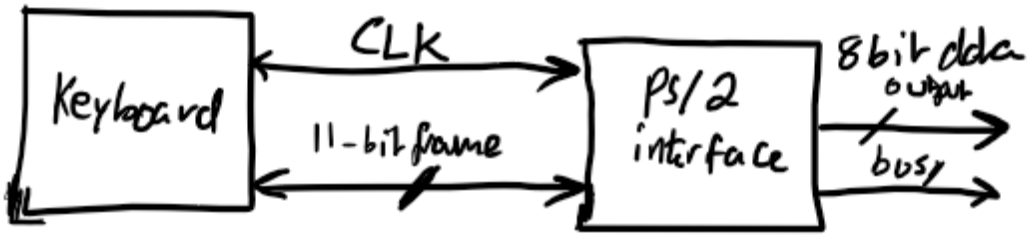


Figure 18 – block diagram showing host and FPGA-based PS/2 interface.

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 an 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

derivation 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.

$$T = \frac{1}{f} = \frac{1}{300bps} = 6.6ms$$

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

$$bit\ period = \frac{T}{2} = 3.333ms$$

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

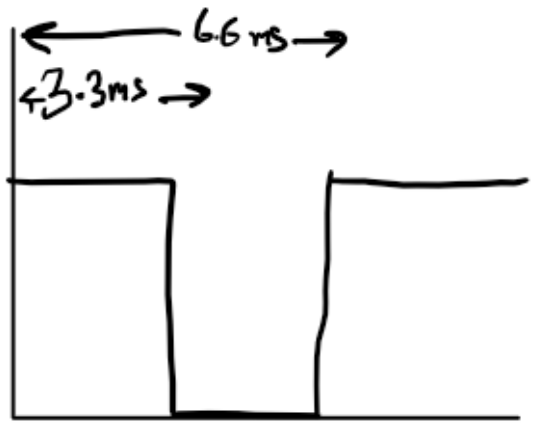


Figure 19 – diagram showing time period (6.6ms) and bit period (3.3ms).

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)

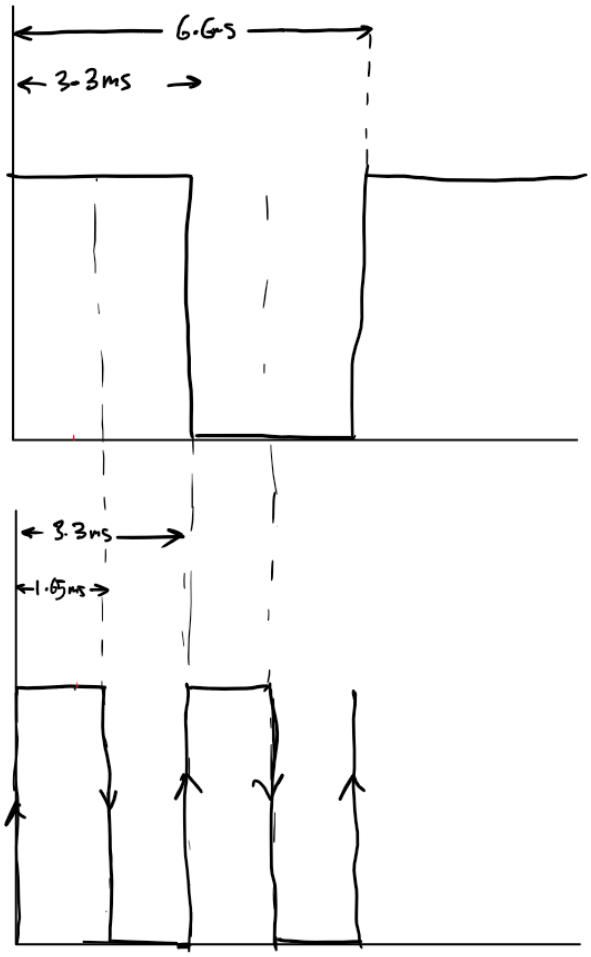


Figure 20 – Top: example UART data at 300 baud. Bottom: sampling clock with time period = half bit period.

Thus, the sampling frequency is:

$$f_{sampling} = \frac{1}{T_{sampling}} = \frac{1}{bit\ period} = \frac{1}{3.333ms} = 300Hz = baud\ rate$$

Thus it can be said that for a desired baud rate, b, the sampling frequency, $f_{sampling}$ is equal to the baud rate.

$$\boxed{\therefore \boldsymbol{f_{sampling} = b}}$$

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.

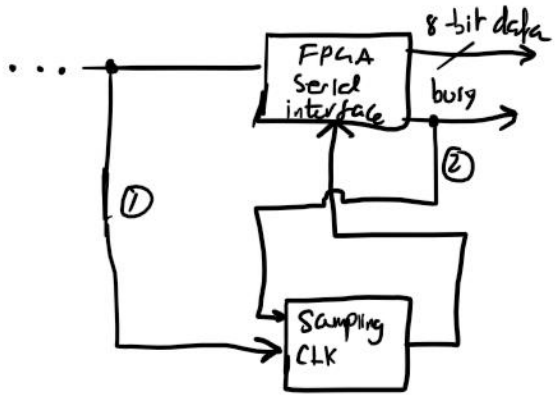


Figure 21 – connection diagram between RS232 receiver and the sampling clock.

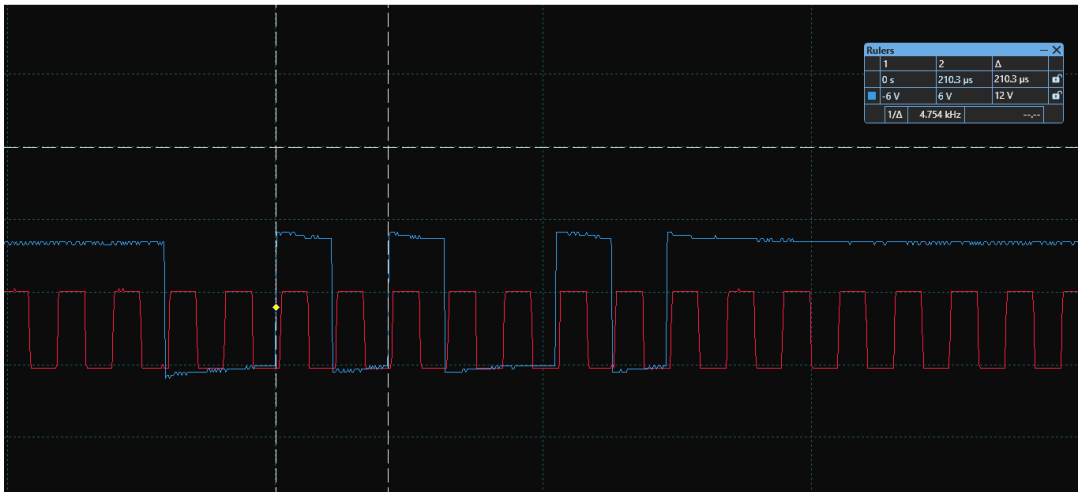


Figure 22 – oscilloscope waveforms. Blue: serial data. Red: raw sampling clock.

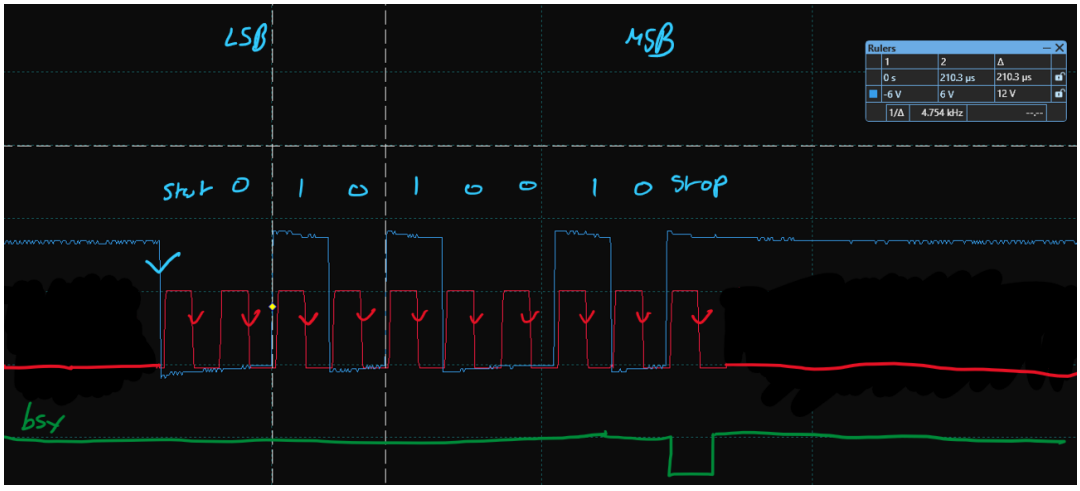


Figure 23 – desired oscilloscope waveforms. Blue: serial data. Red: sampling clock active during frame. Green: busy flag.

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)

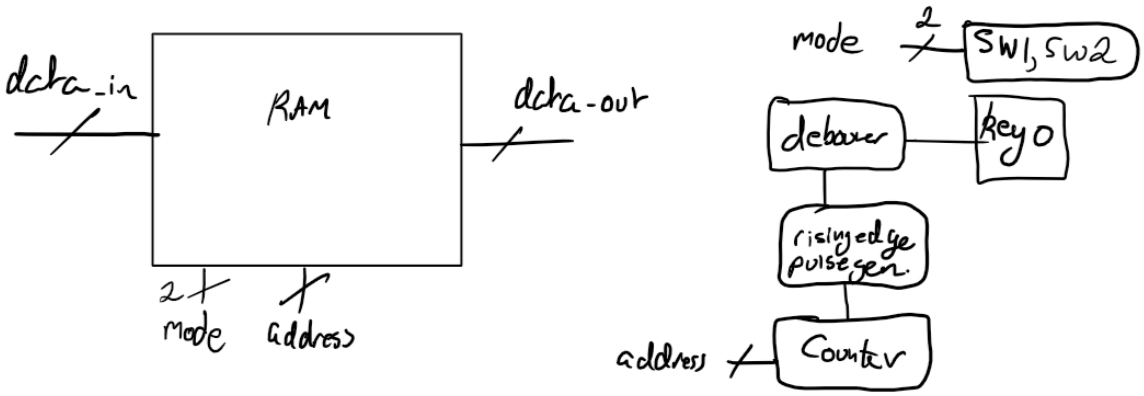


Figure 24 – Instruction RAM connections.

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

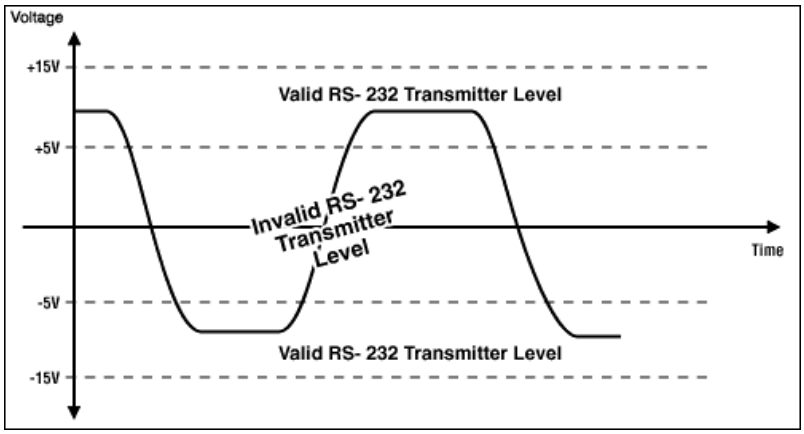


Figure 25 – RS232 voltage levels [13].

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.*

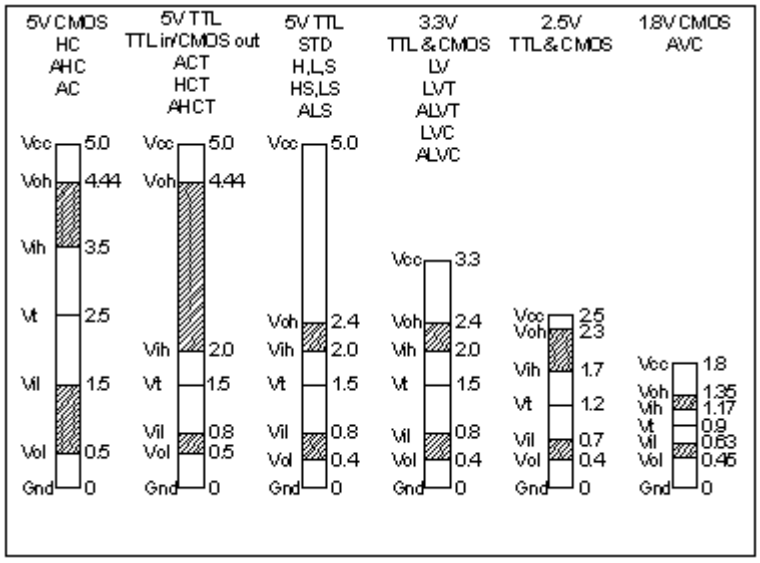


Figure 26 – TTL and CMOS voltage levels [15].

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

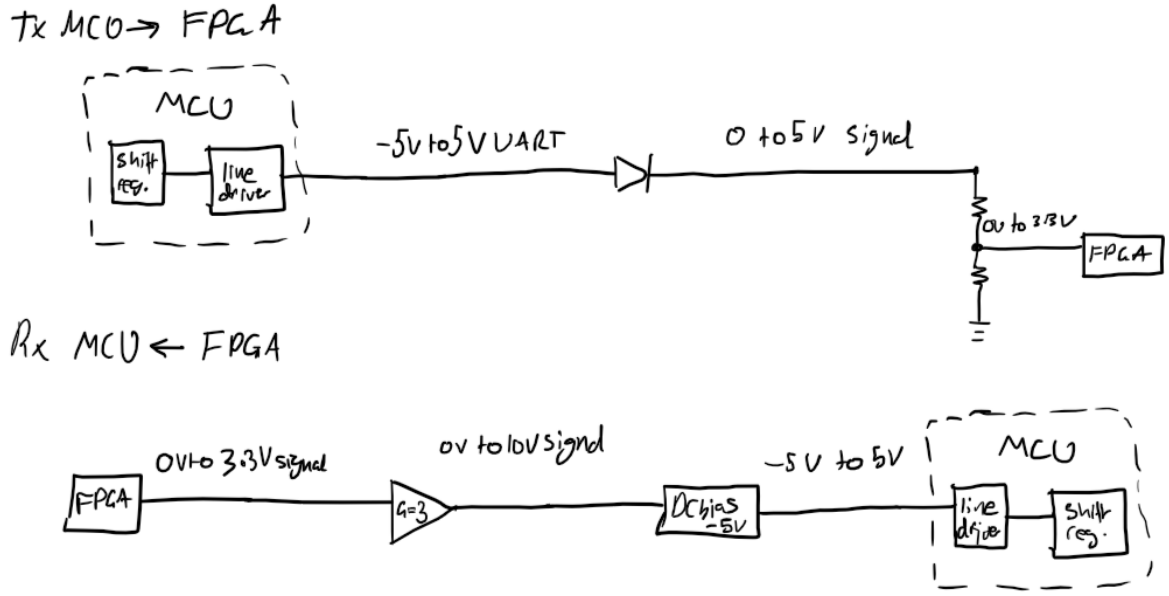


Figure 25 – block diagram of interface circuitry.

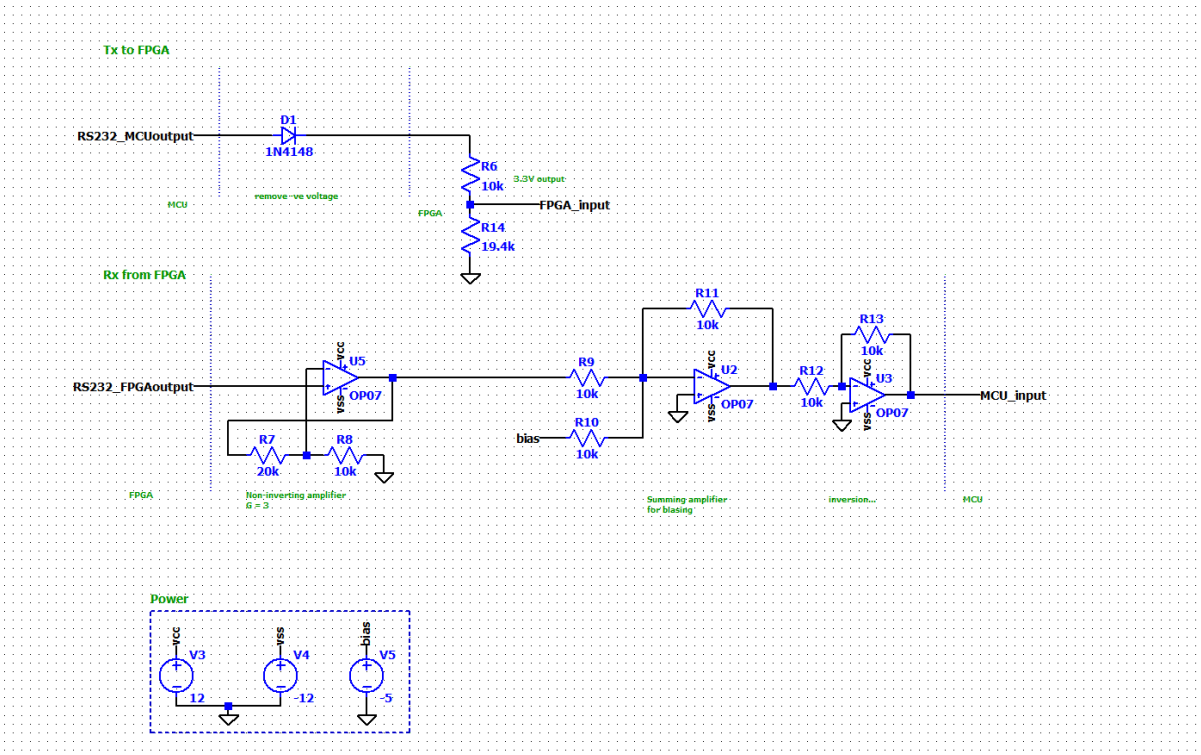


Figure 26 – schematic design of Figure 25.

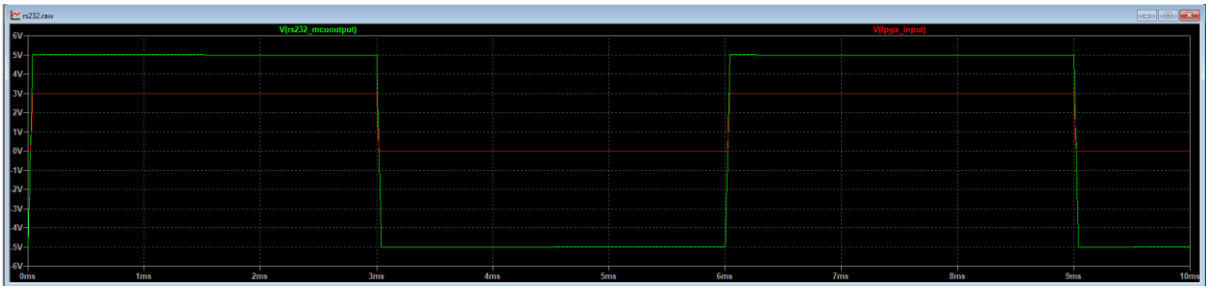


Figure 27 – Tx to FPGA waveforms. Green: line driver output. Red: converted FPGA input.

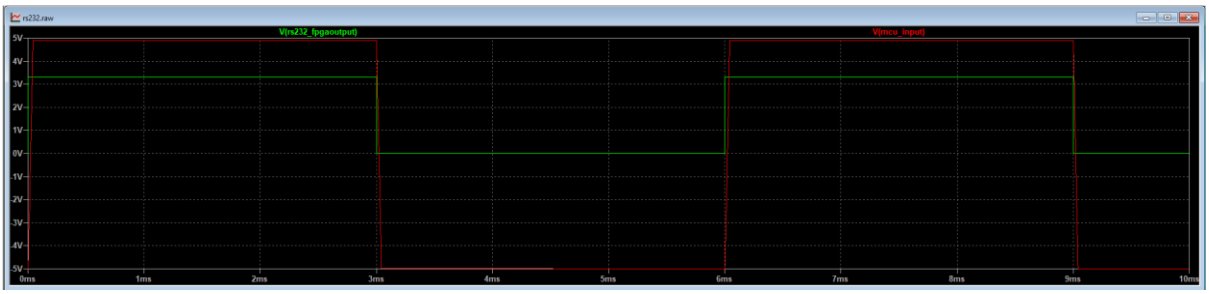


Figure 28 – Rx from FPGA waveforms. Green: FPGA output. Red: converted line driver input.

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.

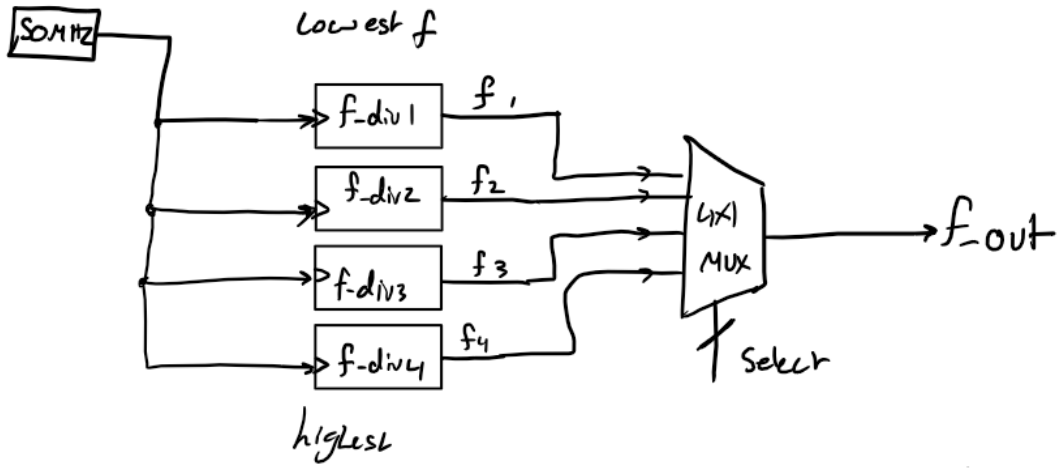


Figure 29 – example of a 4-frequency divider array.

The above design can be easily 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.

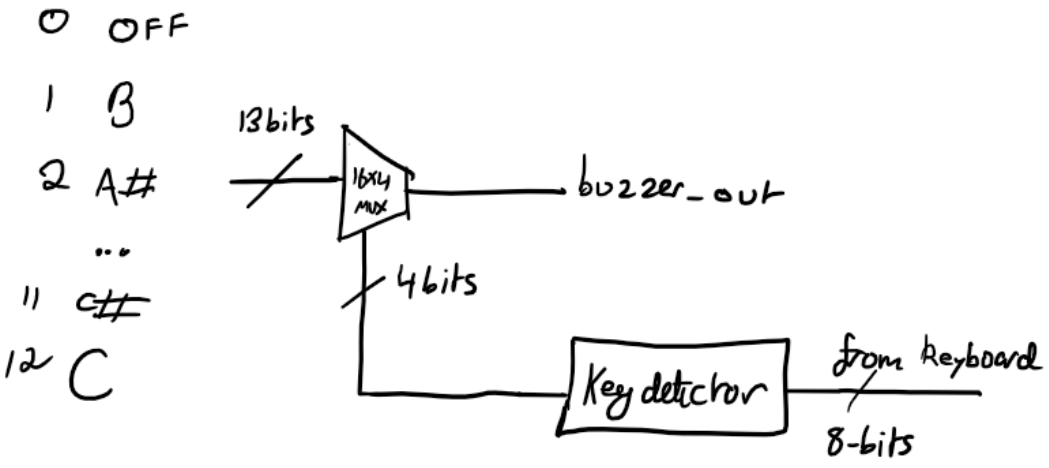


Figure 30 – design incorporating the 13 musical notes.

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 (ISA)

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

Instruction (15 total)	Instruction Code (38-bits)	Description
nop	0000 xxx...xxx	no operation
mov(reg, imm/reg)	0001 x s[0] a[15..12] xxx b[15..0]	move imm/reg
ldr(reg, address/value)	0010 x s[0] a[15..12] xxx b[15..0]	load address/value of var from mem
str(reg/imm, address in mem)	0011 x s[0] a[15..0] b[15..0]	store imm/reg to memory
cmp(reg, imm/reg)	0100 1 s[0] a[15..0] b[15..0]	compare reg with imm/reg
b(address)	0101 xxx...xxx b[15..0]	unconditional branch
bgt(address)	0110 xxx...xxx b[15..0]	branch greater than
blt(address)	0111 xxx...xxx b[15..0]	branch less that
beq(address)	1000 xxx...xxx b[15..0]	branch equal to
add(reg, imm/reg, imm/reg)	1001 s[1..0] a[15..0] b[15..0]	addition
sub(reg, imm/reg, imm/reg)	1010 s[1..0] a[15..0] b[15..0]	subtraction
mul(reg, imm/reg, imm/reg)	1011 s[1..0] a[15..0] b[15..0]	multiplication
lsr(reg, imm)	1100 xx a[15..12] xxx b[15..0]	logical shift to right (div by 2^n)
and(reg, imm/reg, imm/reg)	1101 s[1..0] a[15..0] b[15..0]	logical and
or(reg, imm/reg, imm/reg)	1110 s[1..0] a[15..0] b[15..0]	logical or
mvn(reg, imm/reg)	1111 x s[0] a[15..12] xxx b[15..0]	moving logical not

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

Operation	Extra info
NULL	
reg(a[15..12]) <- b / reg(b[15..12])	top 4 bits of operands 'a' and 'b' is used to select r0-r12; s[1] is don't care; s[0] chooses imm/reg
reg(a[15..12]) <- b / M[b]	s[1] is don't care; s[0] chooses address/value
M[b] <- a / reg(a[15..12])	s[1] is don't care; s[0] chooses imm/reg
a / reg(a[15..12]) - b	sets NZCV flags; s[1] is don't care; s[0] chooses imm/reg
GOTO b	
GOTO b	Z clear, N and V are the same
GOTO b	N and V are different
GOTO b	Z set
reg(a[15..12]) <- a[11..0] / reg(a[11..8]) + b[11..0] / reg(b[11..8])	top 4 bits of operands 'a' and 'b' is used to select r0-r12; s[1..0] chooses between imm/reg for both operands
reg(a[15..12]) <- a[11..0] / reg(a[11..8]) - b[11..0] / reg(b[11..8])	
reg(a[15..12]) <- a[11..0] / reg(a[11..8]) * b[11..0] / reg(b[11..8])	
reg(a[15..12]) <- reg(a[11..8]) >> b	s is don't care
reg(a[15..12]) <- a[11..0] / reg(a[11..8]) & b[11..0] / reg(b[11..8])	
reg(a[15..12]) <- a[11..0] / reg(a[11..8]) | b[11..0] / reg(b[11..8])	
reg(a[15..12]) <- ! b[11..0] / !reg(b[15..12])	

Figure 33 – ISA details.

Where:

- **a** and **b:** operand bits.
- **s:** selector bits.
- **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 AL instructions, 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.

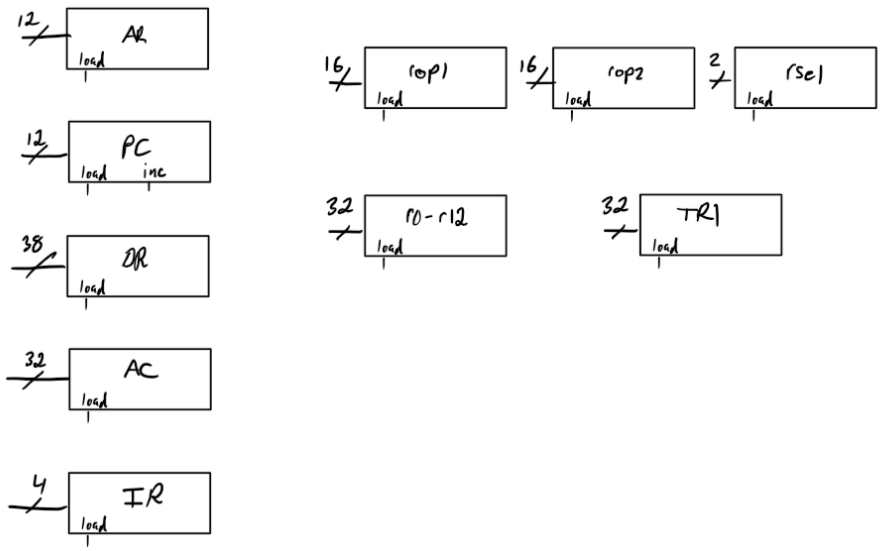


Figure 34 – CPU internal registers, along with their widths.

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

```
fetch1 : AR <- PC
fetch2 : DR <- M; PC <- PC + 1
fetch3 : IR <- DR[37..34]; rsel <- DR[33..32]; rop1 <- DR[31..16]; rop2 <- DR[15..0]
```

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 following 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.

```
example_state1 : reg1 <- value; reg2 <- reg1
```

Figure 35 – example of an illegal state.

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

```
example_state1 : reg1 <-  value
example_state2 : reg2 <- reg1
```

Figure 35 – example state now legal!

2. No Operation (NOP) – 1 clock cycle

```
nop1 : (do nothing)
```

Figure 36 – the NOP operation.

This instruction is usually used for timing synchronisation if required.

3. Move (MOV) – 1/2 clock cycles

```
rsel[0] == 0
mov1 : r <- rop2

rsel[0] == 1
ALTmov1 : TR <- r (rop2)
ALTmov2 : r (rop1) <- TR
```

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.

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

```
rsel[0] == 0
ldr1 : AR <- rop2
ldr2 : r <- AR

rsel[0] == 1
ALTldr1 : AR <- rop2
ALTldr2 : r <- AR
ALTldr3 : DR <- M
ALTldr4 : r <- DR
```

Figure 37 – the LDR operation.

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

5. Store Register (STR) – 4/4 clock cycles

```
rsel[0] == 0
str1 : AR <- rop2
str2 : TR <- r
str3 : DR <- TR
str4 : M <- DR

rsel[0] == 1
ALTstr1 : AR <- rop2
ALTstr2 : TR <- rop1
ALTstr3 : DR <- TR
ALTstr4 : M <- DR
```

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.

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

```
cmp1 : AC <- r - data
```

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).

7. Unconditional Branch (B) – 1 clock cycle

```
b1    : PC <- rop2
```

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.

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

```
bgt1 : PC <- rop2 (Z clear, N and V are the same)
```

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.

Table 5.1. Condition code suffixes

Sign	Suffix	Meaning	Flags
	EQ	Equal	Z = 1
	NE	Not equal	Z = 0
	CS	Carry set (identical to HS)	C = 1
	CC	Carry clear (identical to LO)	C = 0
	MI	Minus or negative result	N = 1
	PL	Positive or zero result	N = 0
	VS	Overflow	V = 1
	VC	Now overflow	V = 0
	AL	Always. This is the default	-
Unsigned	HI	Higher	C = 1 AND Z = 0
	HS	Higher or same	C = 1
	LS	Lower or same	C = 0 OR Z = 1
	LO	Lower (identical to CC)	C = 0
Signed	GT	Greater than	Z = 0 AND N = V
	GE	Greater than or equal	N = V
	LE	Less than or equal	Z = 1 OR N != V
	LT	Less than	N != V

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

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

```
blt1 : PC <- rop2 (N and V are different)
```

Figure 44 – the BLT operation.

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

```
beq1 : PC <- rop2 (Z set)
```

Figure 45 – the BEQ operation.

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

```
add1 : AC <- data + data
add2 : r <- AC
```

Figure 46 – the ADD operation.

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

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

```
sub1 : AC <- data - data
sub2 : r <- AC
```

Figure 47 – the SUB operation.

13. Multiply (MUL) – 2 clock cycles

```
mul1 : AC <- data * data
mul2 : r <- AC
```

Figure 48 – the MUL operation.

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

```
lsr1 : AC <- r >> rop2
lsr2 : r <- AC
```

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 way (little FPGA resources). Since lsr N is equivalent to dividing by 2^n.

15. Logical AND (AND) – 2 clock cycles

```
and1 : AC <- data & data
and2 : r <- AC
```

Figure 50 – the AND operation.

16. Logical OR (OR) – 2 clock cycles

```
or1  : AC <- data | data
or2  : r <- AC
```

Figure 51 – the OR operation.

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

```
mvn1 : AC <- !data
mvn2 : r <- AC
```

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.

```
rsel[0] == 0
ldr1 : AR <- rop2
ldr2 : r <- AR

rsel[0] == 1
ALTldr1 : AR <- rop2
ALTldr2 : r <- AR       // not needed; could optimise out
ALTldr3 : DR <- M
ALTldr4 : r <- DR
```

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.

```
rsel[0] == 0
str1 : AR <- rop2
str2 : TR <- r          // could optimise this out
str3 : DR <- TR
str4 : M <- DR

rsel[0] == 1
ALTstr1 : AR <- rop2
ALTstr2 : TR <- rop1
ALTstr3 : DR <- TR
ALTstr4 : M <- DR
```

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.

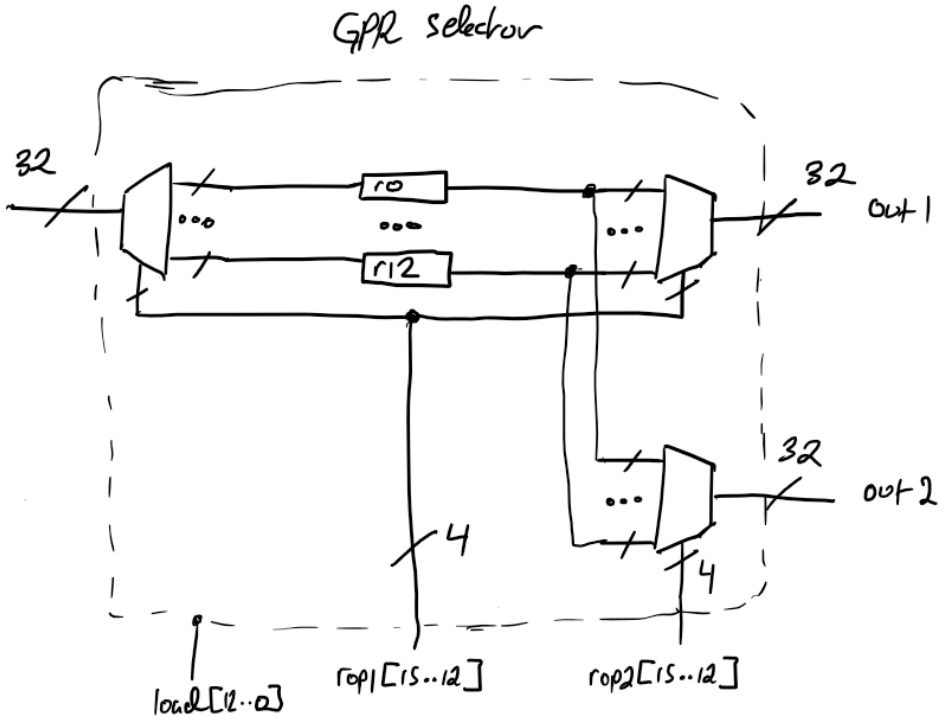


Figure 55 – GPR Selector module; makes use of a DEMUX-MUX pair.

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 AL instructions – 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).

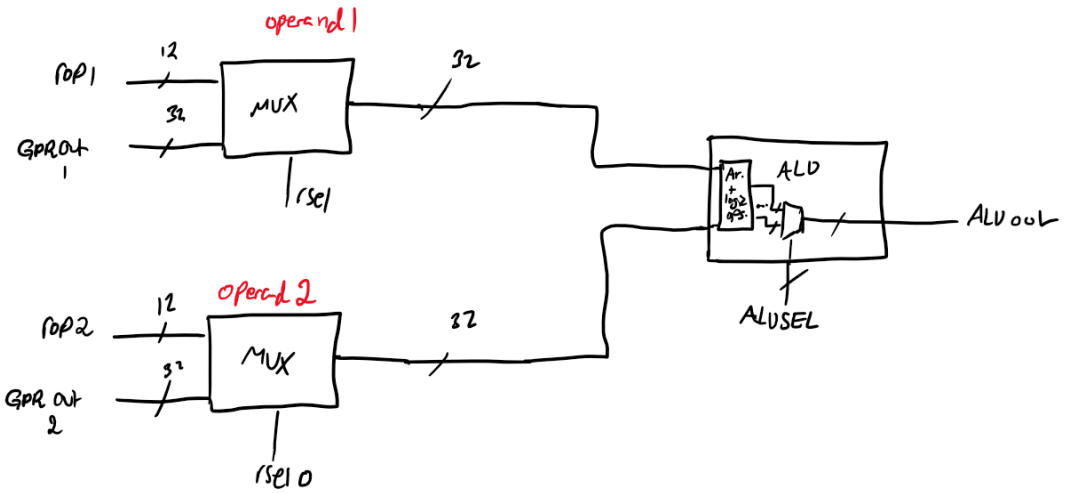


Figure 56 – ALU circuitry. *GPROUT1 and GPROUT2 represent the outputs of two GPR registers.*

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.
    a. ***ADD instruction:*** *if an overflow occurred.*
    b. ***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.
    a. ***ADD instruction:*** *the addition of two positive numbers, and the result is negative; sign bit (MSB) changes to a 1.*
    b. ***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.

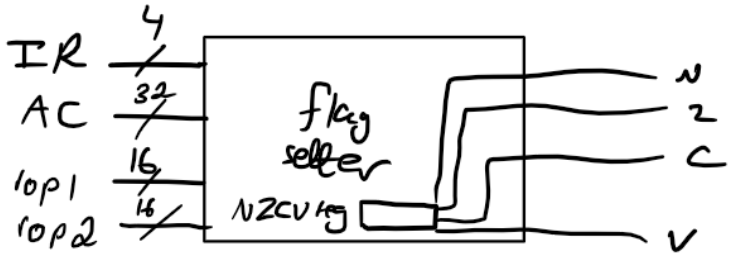


Figure 57 – *flags_setter* module.

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:

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

```
// counter is incremented by CU via inc input! //

// 16 possible opcodes may be input
// indicating START state of execution routine
// 0  : nop
// 1  : mov
// 2  : ldr
// 3  : str
// 4  : cmp
// 5  : b
// 6  : bgt
// 7  : blt
// 8  : beq
// 9  : add
// 10 : sub
// 11 : mul
// 12 : lsr
// 13 : and
// 14 : or
// 15 : mvn
```

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

```
// counter mapping is as follows (40 states - * indicates start state of routine)
// fetch1   : 0
// fetch2   : 1
// fetch3   : 2
// nop1     : 3*
// mov1     : 4*
// ALTmov1  : 5*
// ALTmov2  : 6
// ldr1     : 7*
// ldr2     : 8
// ALTldr1  : 9*
// ALTldr2  : 10
// ALTldr3  : 11
// ALTldr4  : 12
// str1     : 13*
// str2     : 14
// str3     : 15
// str4     : 16
// ALTstr1  : 17*
// ALTstr2  : 18
// ALTstr3  : 19
// ALTstr4  : 20
// cmp1     : 21*
// b1       : 22*
// bgt1     : 23*
// blt1     : 24*
// beq1     : 25*
// add1     : 26*
// add2     : 27

// sub1     : 28*
// sub2     : 29
// mul1     : 30*
// mul2     : 31
// lsr1     : 32*
// lsr2     : 33
// and1     : 34*
// and2     : 35
// or1      : 36*
// or2      : 37
// mvn1     : 38*
// mvn2     : 39
```

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

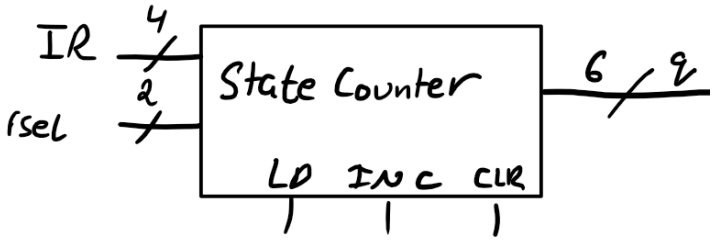


Figure 60 – *State Counter* module. *6 bits to allow counting to 40.*

2. 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.

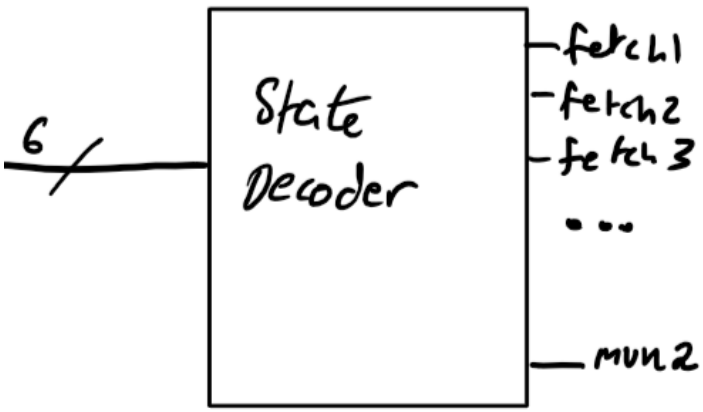


Figure 62 – *State Decoder* module.

* 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)

3. The Control Signal Logic Controller

Finally, the module that does the “control” part in the Control Unit. This massive 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.**

```
Counter lines

LD  = fetch3

INC = fetch1 | fetch2 | ALTmov1 | ldr1 | ALTldr1 | ALTldr2 | ALTldr3 | str1 | str2 | str3 |
    | ALTstr1 | ALTstr2 | ALTstr3 | add1 | sub1 | mul1 | lsr1 | and1 | or1 | mvn1

CLR = nop1 | mov1 | ALTmov2 | ldr2 | ALTldr4 | str4 | ALTstr4 |
    | cmp1 | b1 | bgt1 | blt1 | beq1 | add2 | sub2 | mul2 | lsr2 | and2 | or2 | mvn2
```

Figure 63 – *State Counter* control signals.

```
Load lines (LHS)

RSELLOAD = fetch3
ROP1LOAD = fetch3
ROP2LOAD = fetch3
TRLOAD   = ALTmov1 | str2 | ALTstr2
ARLOAD   = fetch1 | ldr1 | ALTldr1 | str1 | ALTstr1
PCLOAD   = b1 | bgt1 | blt1 | beq1
DRLOAD   = fetch2 | ALTldr3 | str3 | ALTstr3
ACLOAD   = cmp1 | add1 | sub1 | mul1 | lsr1 | and1 | or1 | mvn1
IRLOAD   = fetch3
GPRLOAD  = mov1 | ALTmov2 | ldr2 | ALTldr2 | ALTldr4 | add2 | sub2 | mul2 | lsr2 | and2 | or2 | mvn2
```

Figure 64 – register load control signals.

```
Increment lines

PCINC = fetch2
```

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

```
Memory load line

MEMLOAD = str4 | ALTstr4
```

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

```
System bus access (RHS) (number is the SYSTEMBUSSEL value)

PC   = fetch1                                                                  0
DR   = fetch3 | ALTldr4 | str4 | ALTstr4                                       1
AR   = ldr2 | ALTldr2                                                          2
AC   = add2 | sub2 | mul2 | lsr2 | and2 | or2 | mvn2                           3
MEM  = fetch2 | ALTldr3                                                        4
TR   = ALTmov2 | str3 | ALTstr3                                                5
rop1 = ALTstr2                                                                 6
rop2 = mov1 | ldr1 | ALTldr1 | str1 | ALTstr1 | b1 | bgt1 | blt1 | beq1       7
GPR1 = ALTmov1 | str2                                                         8
```

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

```
ALU has 7 operations; ALUSEL is 3-bits (number is the ALUSEL value)
ALUSEL asserted when AC is loaded!

0 : add1
1 : sub1
2 : mul1
3 : lsr1
4 : and1
5 : or1
6 : mvn1
```

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

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.

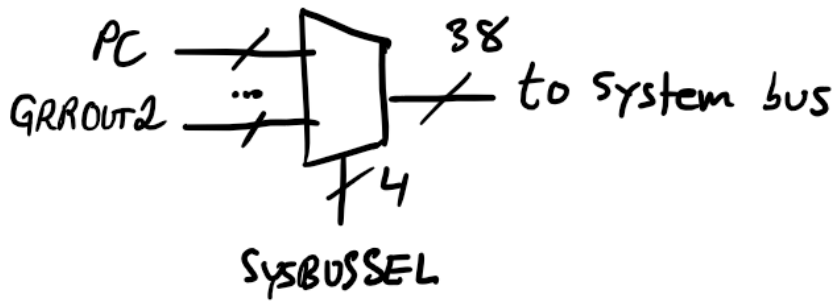


Figure 70 – *System Bus* used to avoid contention.

7.5.10 Final CPU Datapath (wiring)

The final architecture of the CPU is shown below.

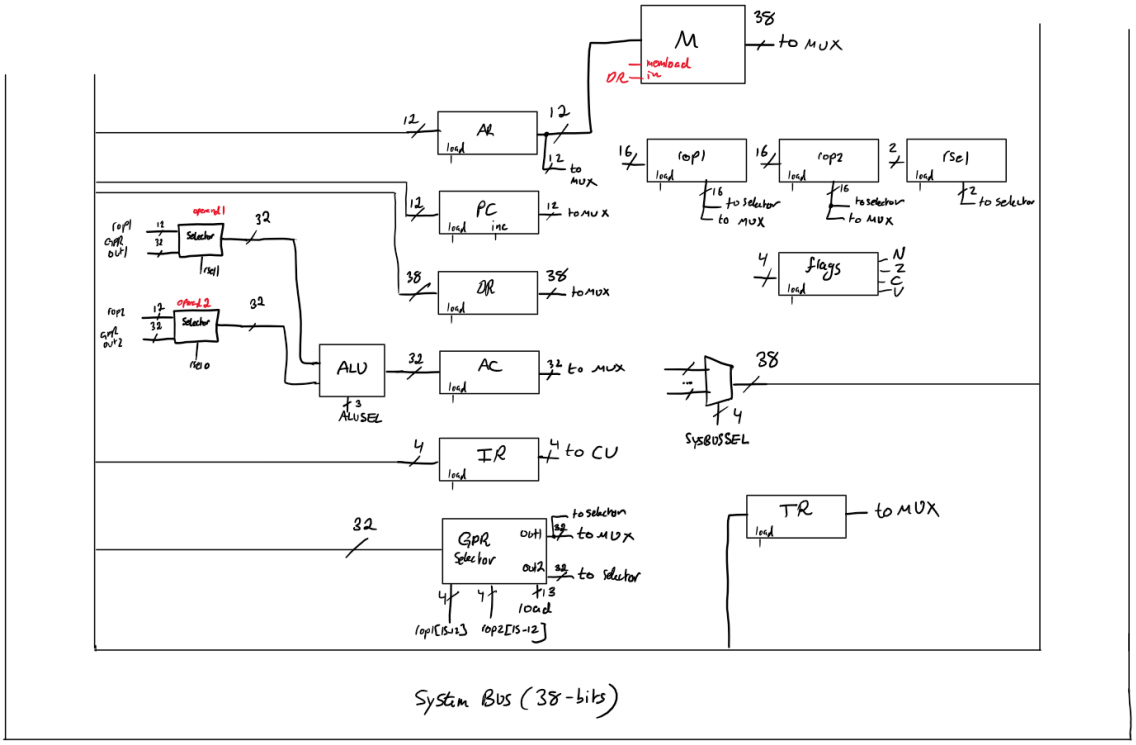


Figure 71 – Final CPU architecture.

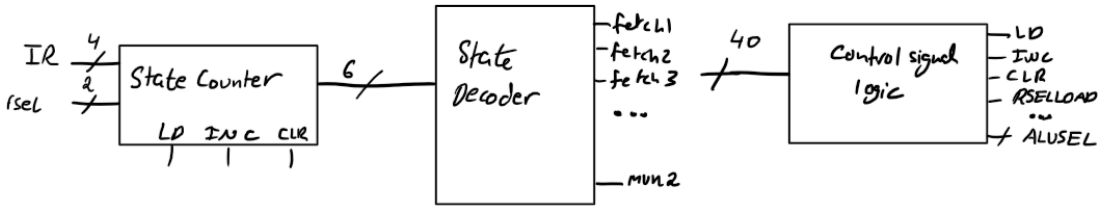


Figure 72 – The Control Unit.

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 registers *(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.

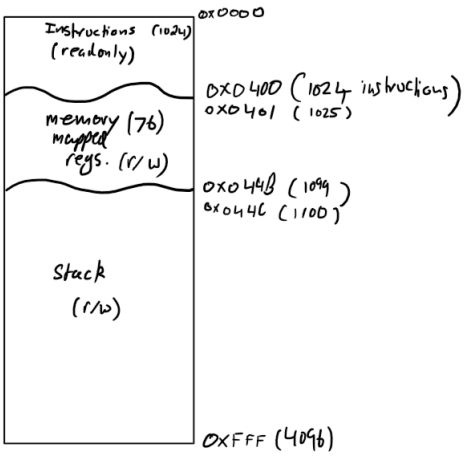


Figure 73 – The Memory Map.

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.

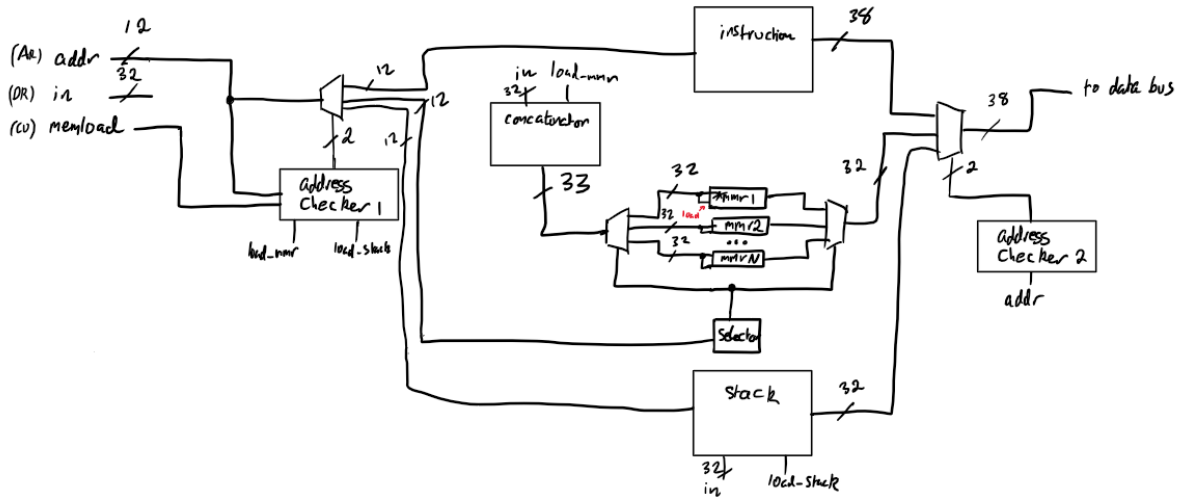


Figure 74 – memory control circuitry. *3 memories can be seen (top to bottom): the instruction memory, the MMR bank, and the stack memory.* ***The MEMLOAD signal is asserted during CPU states str4 and ALTstr4*.**

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.

- **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.

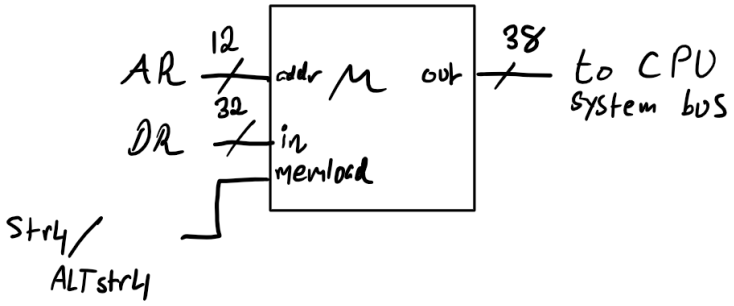


Figure 75 – external CPU memory module *(MMR outputs not shown).*

```
List of memory mapped registers:

0x401 : 9-bit FPGA led (r/w)
0x402 : 8-bit keyboard data (read only)
0x403 : 8-bit keyboard audio data (r/w)
0x404 : row (read only)
0x405 : column (read only)
0x406 : 4-bit keyboard sprite control data (r/w)
0x407 : 1-bit selector (r/w)
0x408 : 12-bit colour data (r/w)
```

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 10 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.

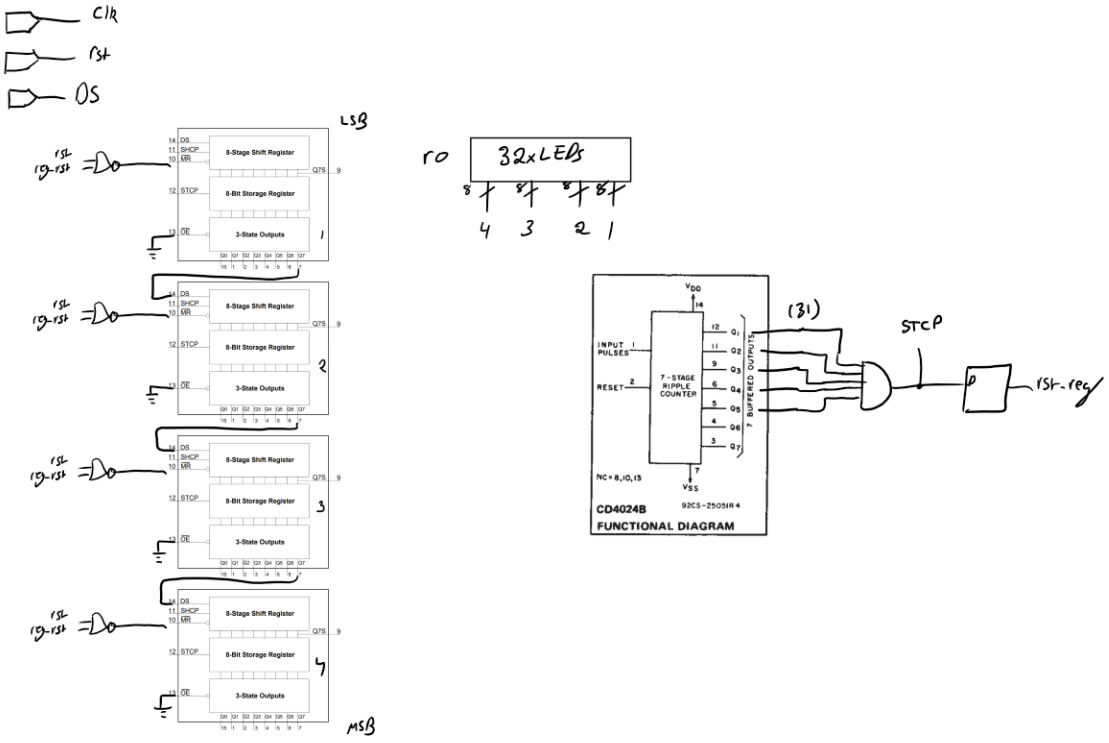


Figure 77 – example circuitry showing 1 LED controller (GPR r0 is shown above). The 8-bit shift register ICs are 74HC595 chips. The clock, reset and serial data (DS) inputs were to come from the FPGA.

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

```
one led controller:

4 x HC595               (shift)
1 x LS04                (inverter)
2 x HC08                (and)
1 x LS74                (ff)
4 x HC573               (latch)
```

Figure 78 – BOM for one LED controller.

```
assuming 13 GPRs + 10 MMRs = 23 devices:

92 x HC595              (shift)
23 x LS04               (inverter)
46 x HC08               (and)
23 x LS74               (ff)
92 x HC573              (latch)
```

Figure 79 – BOM for 23 LED controllers.

```
in addtion to:

1  x CD4075             (or)
1  x CD4024             (counter)
1  x CD4001             (nor)
10 x LS04               (inverter)
2  x LS74               (ff)
4  x HC138              (1x8 demux)
1  x HC139              (1x4 demux)
```

Figure 80 – BOM for “glue circuitry”.

```
Total:

1 x CD4001              (nor)xx
92 x HC595              (shift)xx
1 x CD4075              (or)xx
1 x CD4024              (counter)xx
33 x LS04               (inverter)xx
46 x HC08               (and)xx
25 x LS74               (ff)xx
92 x HC573              (latch)xx
4  x HC138              (1x8 demux)xx
1  x HC139              (1x4 demux)xx

13 GPRs * 32 = 416 red LEDsx
10 MMRs * 32 = 320 yellow LEDsx
MOSFET driver (3.3V -> 5V)x
```

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 (large footprint), 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.*

8.1 The VGA Controller

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

```
// from 0 to 639
if (count_reg < 11'd639) begin

    rgb_en <= 1;
    hsync <= 1;

end


// from 640 to 655
if (count_reg > 11'd638 && count_reg < 11'd655) begin

    rgb_en <= 0;
    hsync <= 1;


end


// from 656 to 751
if (count_reg > 11'd654 && count_reg < 11'd751) begin

    rgb_en <= 0;
    hsync <= 0;


end

// from 752 to 799
if (count_reg > 11'd750 && count_reg < 11'd799) begin

    rgb_en <= 0;
    hsync <= 1;

end

if (count_reg == 11'd800) begin

    count_reg <= '0;
    rgb_en <= 1;
    hsync <= 1;


end
```

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.

```
always_ff @(posedge clk) begin

    if (rst) begin

        count_reg <= 'd0;
        hsync <= 1;
        rgb_en <= 1;

    end

    else begin

        count_reg <= count_reg + 1;

    end
```

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

```
always_ff @(posedge clk) begin

    if (rst) begin

        count_reg <= 'd0;
        vsync <= 1;
        rgb_en <= 1;

    end

    else if (column_reg == 11'd800) begin

        count_reg <= count_reg + 1;

    end
```

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.

```
always_comb begin

    row_reg = row;
    column_reg = column;


    r = colour_reg[11:8];
    g = colour_reg[7:4];
    b = colour_reg[3:0];

end
```

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.

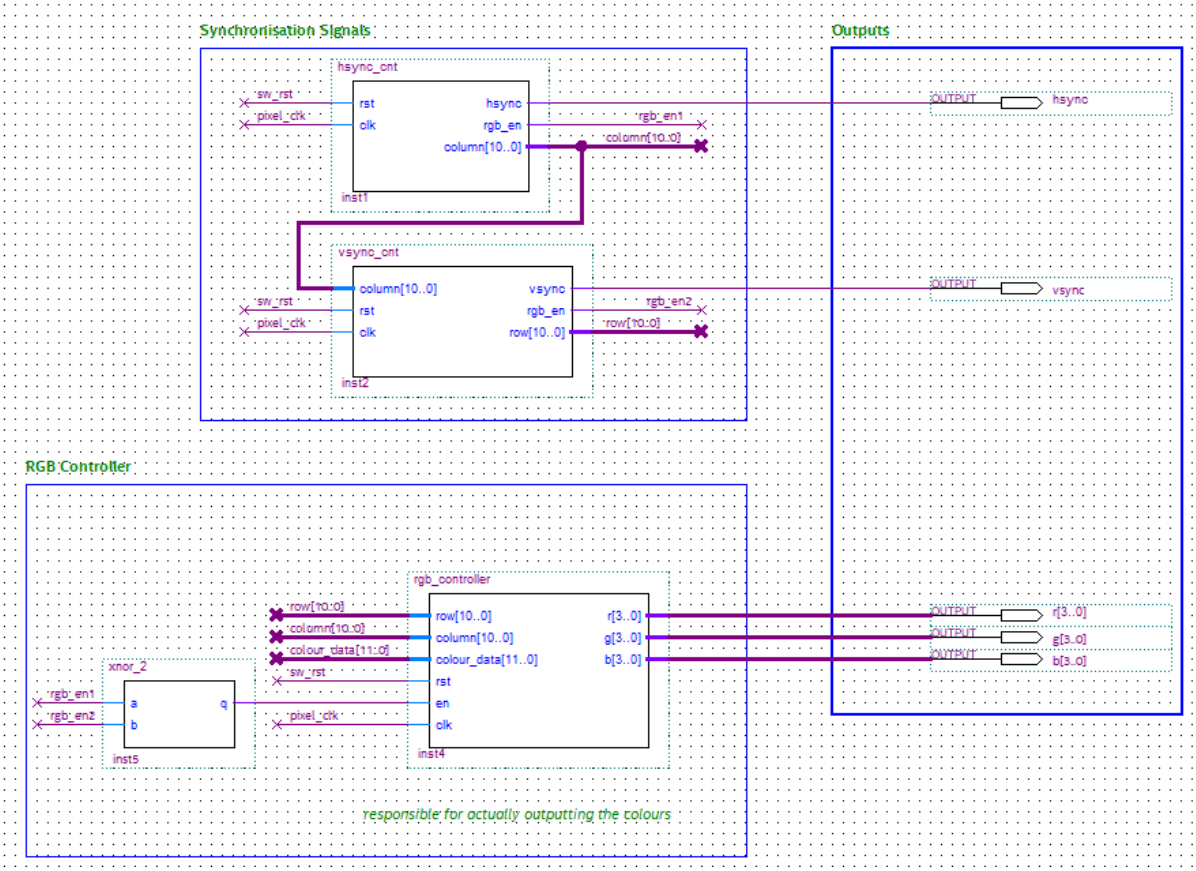


Figure 85 – VGA Controller wiring.

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

```
// coordinates (row,column)
// (100,100) (100,101)
// (101,100) (101,101)

// colours
// red green
// green red
// NOTE: first memory location is the lowest address
logic [11:0] memory [0:2**M-1] = '{

    12'hF00, 12'h0F0,
    12'h0F0, 12'hF00

};
```

Figure 86 –sprite ROM initialisation holding the colour data.

```
// output logic
always_comb begin

    case({row_reg, column_reg})

    {11'sd100 + offset_row_reg, 11'sd100 + offset_column_reg} : begin

        q = memory[0];

    end

    {11'sd100 + offset_row_reg, 11'sd101 + offset_column_reg} : begin

        q = memory[1];

    end

    {11'sd101 + offset_row_reg, 11'sd100 + offset_column_reg} : begin

        q = memory[2];

    end

    {11'sd101 + offset_row_reg, 11'sd101 + offset_column_reg} : begin

        q = memory[3];

    end

    default : begin

        q = 12'h000;

    end


    endcase



end
```

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.

```
{1'b0,4'b1000} : begin

    // pixel2_row_pos
    if (pixel_pos[21:11] != 11'sd0) begin

        row_offset_reg <= row_offset_reg - 1;               // up

    end


end

{1'b0,4'b0100} : begin

    // pixel1_row_pos
    if (pixel_pos[10:0] != 11'sd479) begin

        row_offset_reg <= row_offset_reg + 1;               // down

    end
end
```

```
{1'b0,4'b0010} : begin

    // pixel3_column_pos
    if (pixel_pos[32:22] != 11'sd0) begin

        column_offset_reg <= column_offset_reg - 1;         // left

    end

end

{1'b0,4'b0001} : begin

    // pixel4_column_pos
    if (pixel_pos[43:33] != 11'sd639) begin

        column_offset_reg <= column_offset_reg + 1;         // right

    end

end
```

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 to 11 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.

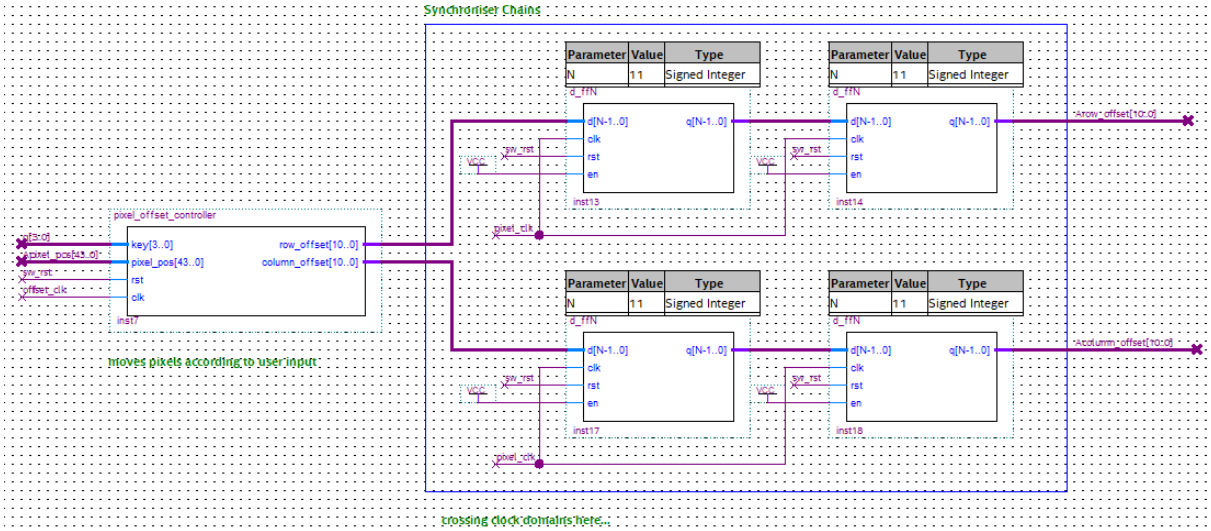


Figure 89 – CDC chains before sending the data to the sprite ROM(s).

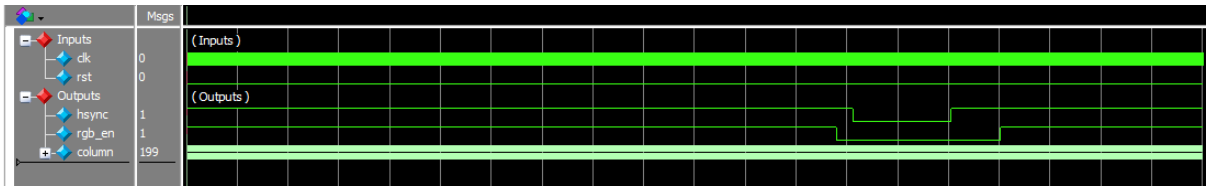


Figure 90 – *hsync_cnt* testbench results.

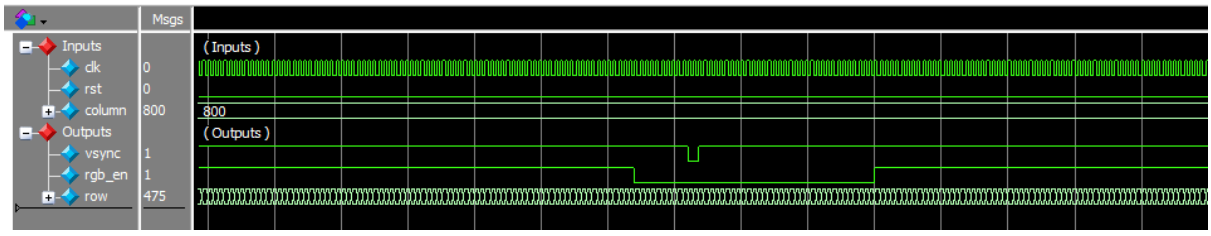


Figure 91 – *vsync_cnt* testbench results. *Note it is only counting when the column input is 800.*

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

```
// coordinates (row,column)
// (100,100) (100,101)
// (101,100) (101,101)
```

Figure 92 – initial sprite coordinates.

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

```
// coordinates (row,column)
// (50,150) (50,151)
// (51,150) (51,151)
```

Figure 93 – offset sprite coordinates.

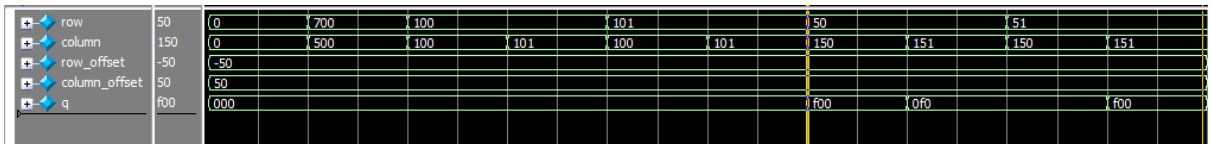


Figure 94 – correct colour data output at the correct row and column coordinates.

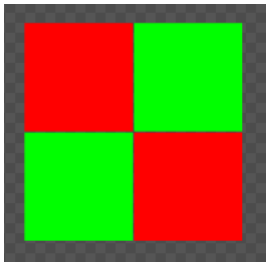


Figure 95 – what the sprite should look like.

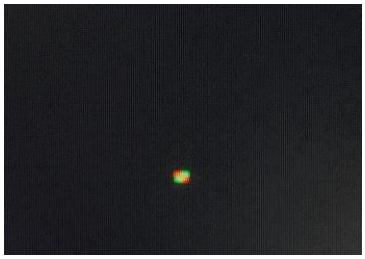


Figure 96 – real life test; what the sprite looks like!

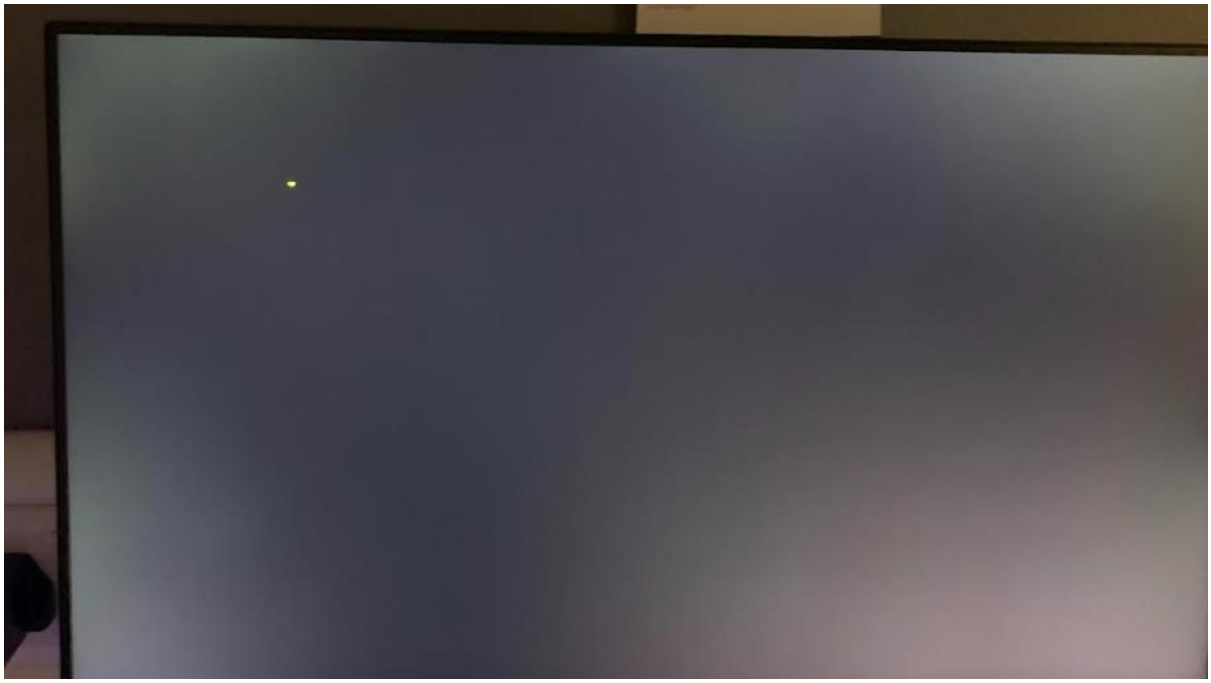


Figure 96 – initial sprite position.

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.

```
else begin

    count_reg <= count_reg + 1;
    bsy <= 1;

    // data frame
    if (count_reg >= 'd1 && count_reg <= 'd8) begin

        data_out[7] <= data_in;

        for (i=6; i>-1; i--) begin

            data_out[i] <= data_out[i+1];


        end


    end

    // busy flag low at the end of transmission
    // can read this safely now
    if (count_reg == 'd10) begin

        bsy <= 0;

    end


    if (count_reg == 'd11) begin

        count_reg <= 'd1;
        data_out <= 'd0; // not necessary, but nice to reset back to zero at EoT


    end


end
```

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 read.

```
always_ff @(posedge rst or negedge bsy) begin

    if (rst) begin

        data_out <= 'd0;

    end

    // only read if bsy flag is low
    else if(!bsy) begin

        data_out <= data_in;

    end

end
```

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.

```
// 50MHz clock input
// if clock used is slower than ps2 clock, you will miss the event!
always_ff @(posedge clk) begin

    // break code upon key release is 2 bytes
    // (0xF0, make code)
    if (data == 8'hF0) begin

        released <= 1;


    end

    // begin timer...
    if (released == 1) begin

        counter <= counter + 1;


    end

    // Tpulse = N/50MHz
    // 5ms timer to reset output back to zero when releasing key
    // holds the flag high for 5ms (longer than data frame)
    // as reset is pulsed between the two bytes, it is CRUCIAL that
    // the reset pulse time > data frame
    if (counter == 'd250000) begin

        released <= 'd0;
        counter <= 'd0;


    end


end
```

Figure 100 – *break_code_detector* module code.

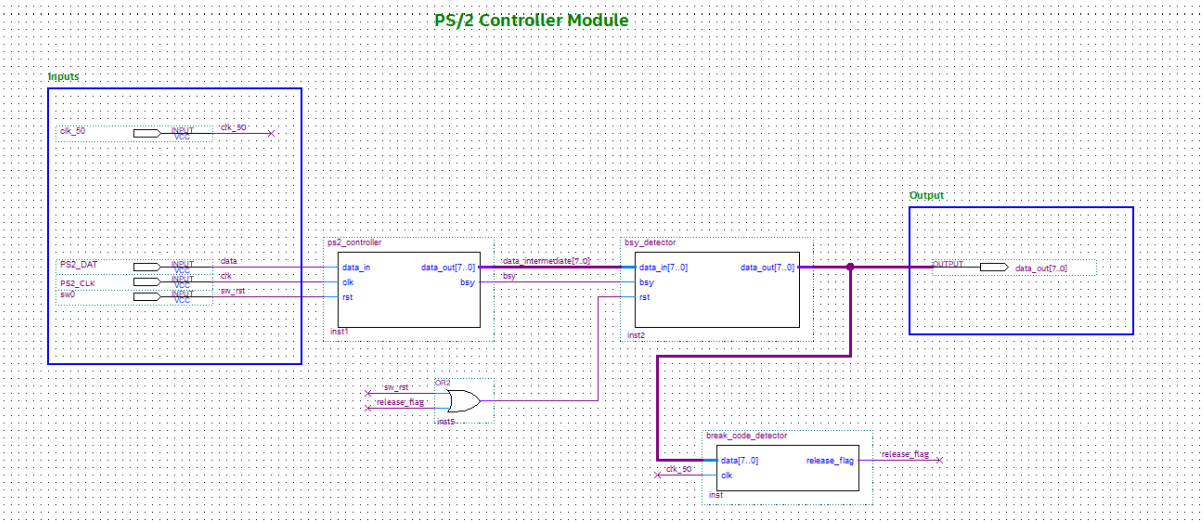


Figure 101 – PS/2 Keyboard Controller wiring. *Note the OR of the two reset signals.*

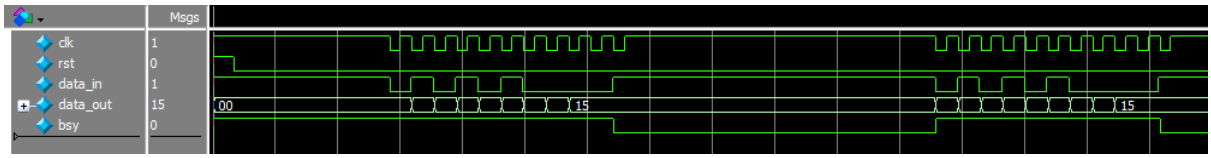


Figure 102 – *ps2_controller* module testbench results. This involved simulating the press of key ‘Q’ (make code 0x15) twice.

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.

```
// if Enter key is pressed

case(enter_pressed)


// key# -> Enter+key
// e.g. A# -> Enter + A

// we have:
// A# (2)
// G# (4)
// F# (6)
// D# (9)
// C# (11)

'd1 : begin
    case(keyboard_data)

    // A#
    8'h1C : begin
        q <= 'd2;

    end

    // G#
    8'h34 : begin
        q <= 'd4;

    end
```

Figure 104 – snippet of the sharp keys lookup table.

```
'd0 : begin
    case(keyboard_data)

    // B
    8'h32 : begin
        q <= 'd1;
    end

    // A
    8'h1C : begin
        q <= 'd3;
    end

    // G
    8'h34 : begin
        q <= 'd5;
    end
```

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

```
default : begin

    q <= 'd0;

end
```

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.

```
// ps2 clock
// so it toggles once per key press
always_ff @(negedge clk) begin

    // Enter - sharp notes toggle
    if (data == 8'h5A) begin

        enter_flag = ~enter_flag;

    end

    // Rshift - octave toggle
    if (data == 8'h59) begin

        rshift_flag = ~rshift_flag;

    end


end
```

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 f_{in}. The output frequency signal, f_{out}, would be hard-wired to the MSB of the counter, **bit N-1**. Leading to the following definition:

$$f_{out} = \frac{f_{in}}{2^N}$$

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:

$$f_{out} = \frac{50\ MHz}{2^{16}} = 762.9395\ Hz$$

And with N=17:

$$f_{out} = \frac{50\ MHz}{2^{17}} = 381.4697\ Hz$$

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, f_{in}. 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.

$$Time\ taken\ to\ reach\ value, T' = \frac{100}{1\ Hz} = 100\ s$$

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

$$Time\ taken\ to\ reach\ value, T' = \frac{100}{2\ Hz} = 50\ s$$

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

$$\therefore \boldsymbol{Time\ taken\ to\ reach\ value, T' = \frac{N}{f_{in}}\ (seconds)}$$

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,

$$Time\ period, T = 2T' = \frac{2N}{f_{in}}$$

$$\therefore \boxed{f_{out} = \frac{1}{T} = \frac{f_{in}}{2N}}$$

With that, a **parameterisable** *fine clock divider* module can be created, where the user can input their desired output frequency, f_{out}, 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:

$$\boxed{N = \frac{f_{in}}{2f_{out}} = \frac{50\ MHz}{2f_{out}}}$$

```
// clk_out = 50MHz/2*N -> N = 50MHz/2*f
// where f = clk_out (desired output frequency)
// solving for (2^25 - 1) = 50MHz/2*f
// -> lowest frequency attainable with 25-bits is 0.745Hz
parameter N=(50000000)/(2*f);
logic [24:0] value = N;

always_ff @(posedge clk_in) begin
    case(rst)
    1 : begin
        counter_reg <= 'd0;
        clk_out <= 0;
    end
    0 : begin
        if (counter_reg != value) begin
            counter_reg <= counter_reg + 1;
        end
        else begin
            clk_out <= ~clk_out;
            counter_reg <= 'd0;
        end
    end
    endcase

end
```

Figure 108 – *fine_clk_divN* module code.

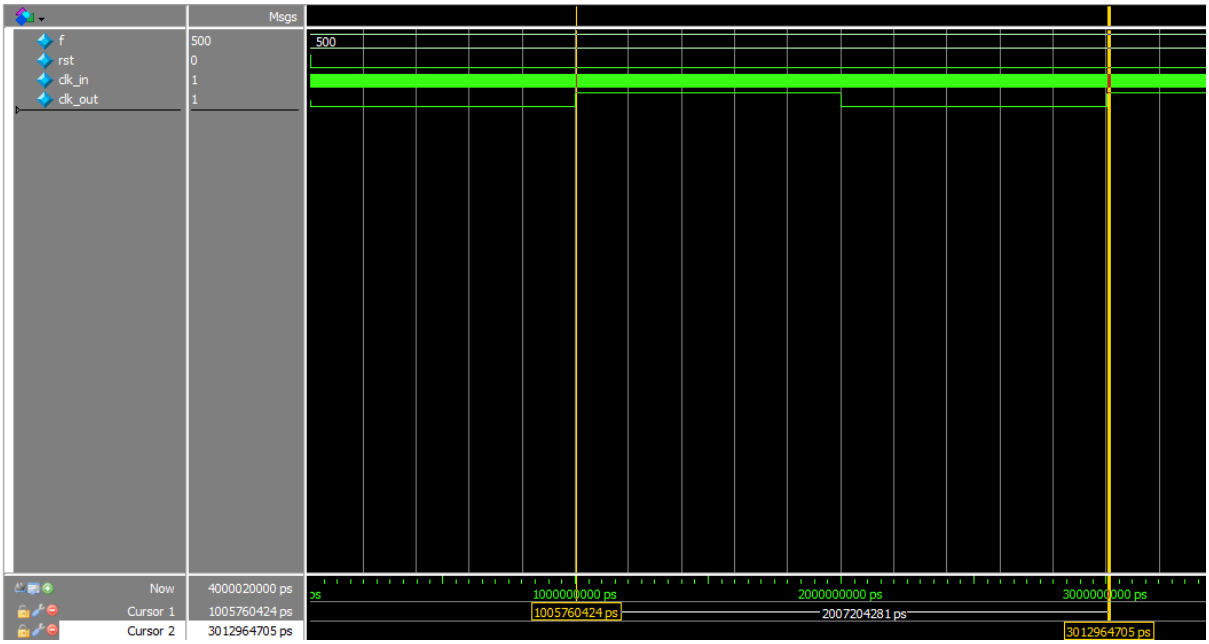


Figure 109 – *fine_clk_divN* testbench waveform with the previously unattainable frequency of f=500 Hz (T=2 ms; see cursor time delta).

To evaluate the accuracy of this clock divider as opposed to the simpler version, a graph of $\boldsymbol{f_{out}(N) = \frac{50}{2N}}$ *is shown below.*

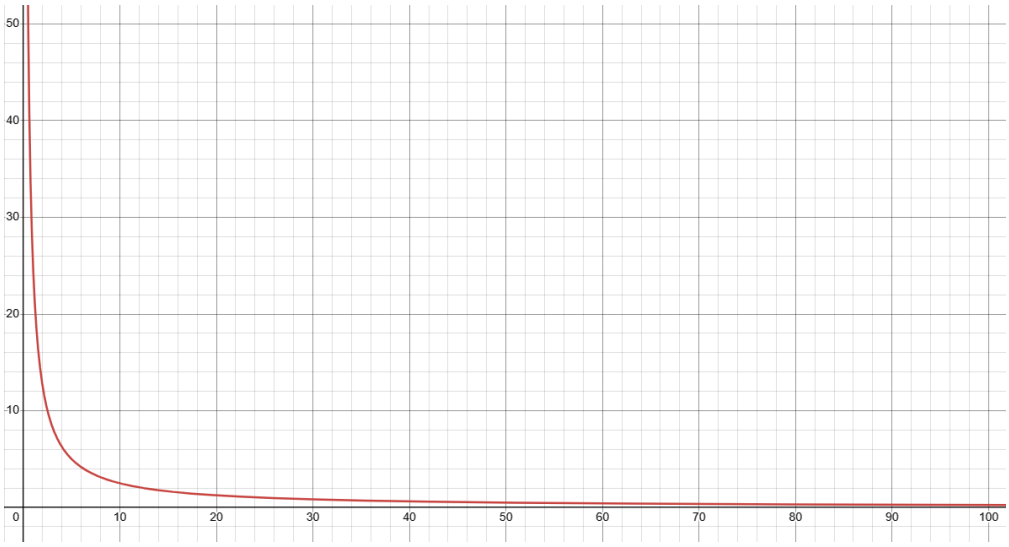


Figure 110 – graph of the relationship between f_{out} and N @ $f_{in} = 50$ (normalised to 1 MHz).

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:

$$N = 50, \qquad f_{out} = \frac{50\ MHz}{2(50)} = 500{,}000\ Hz$$

$$N = 51, \qquad f_{out} = \frac{50\ MHz}{2(51)} = 490{,}196.0784\ Hz$$

$$\therefore \boldsymbol{\Delta f_{out} = 9803.92\ Hz\ \approx 1\ kHz}$$

Large N example:

$$N = 50000, \qquad f_{out} = \frac{50\ MHz}{2(50000)} = 500\ Hz$$

$$N = 50001, \qquad f_{out} = \frac{50\ MHz}{2(50001)} = 499.99\ Hz$$

$$\therefore \boldsymbol{\Delta f_{out} = 0.0099998\ Hz < 1\ Hz}!$$

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.

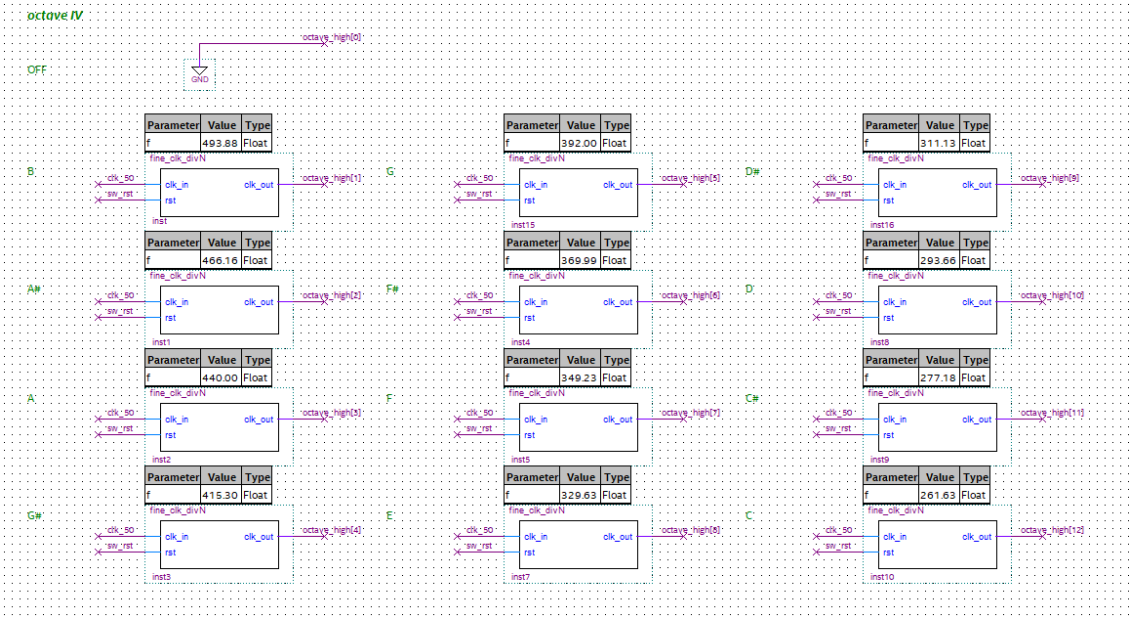


Figure 111 – array of musical frequencies in octave IV (according to Figure 9).

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.

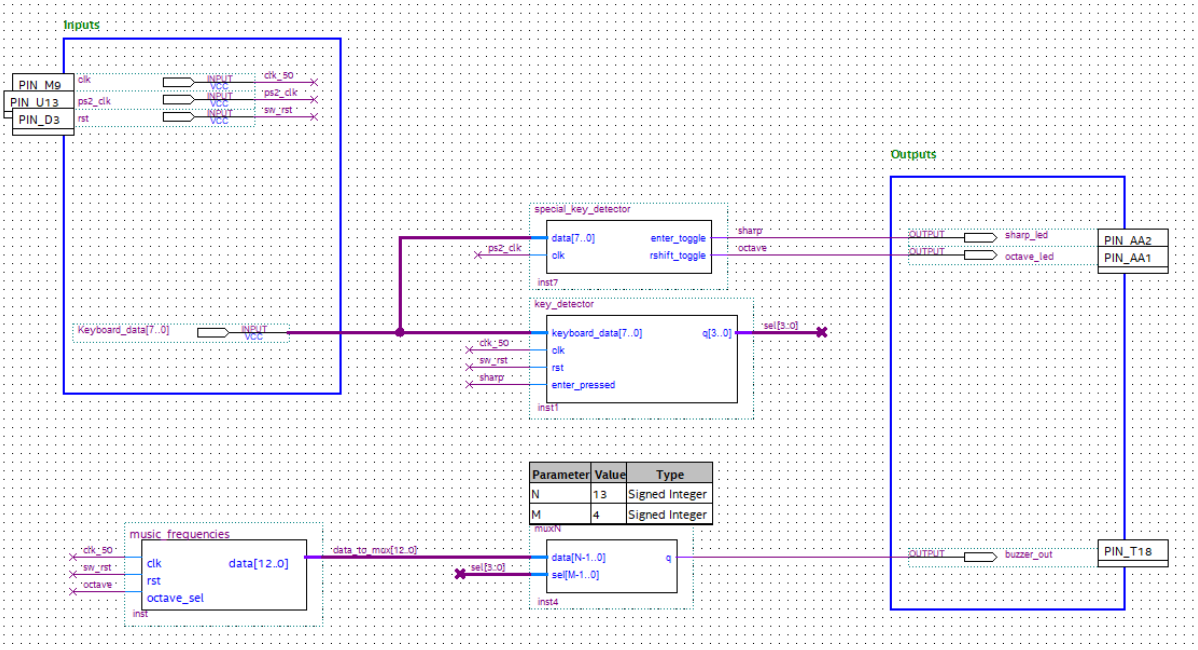


Figure 112 – Audio Controller module wiring.

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

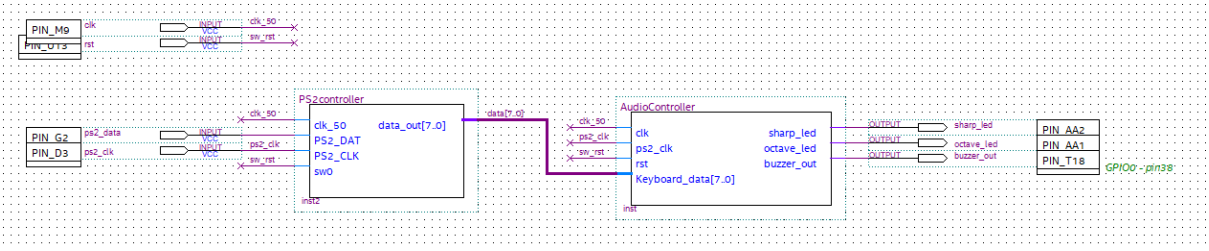


Figure 113 – Audio Controller module wiring.

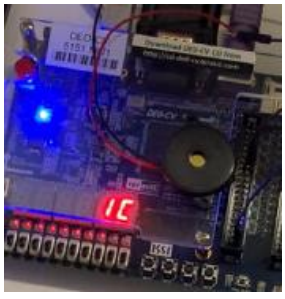


Figure 114 – PWM buzzer playing note A when key ‘A’ (make code 0x1C) is pressed.

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.

```
// no longer idle
// this takes precedence
if (!sense) begin

    transmission_state <= 1;    // sot

end

else if (!bsy) begin

    bsy_posedge_waiting <= 1;


end


if (bsy_posedge_waiting == 1 && bsy == 1) begin

    transmission_state <= 0;    // eot
    bsy_posedge_waiting <= 0;



end
```

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.

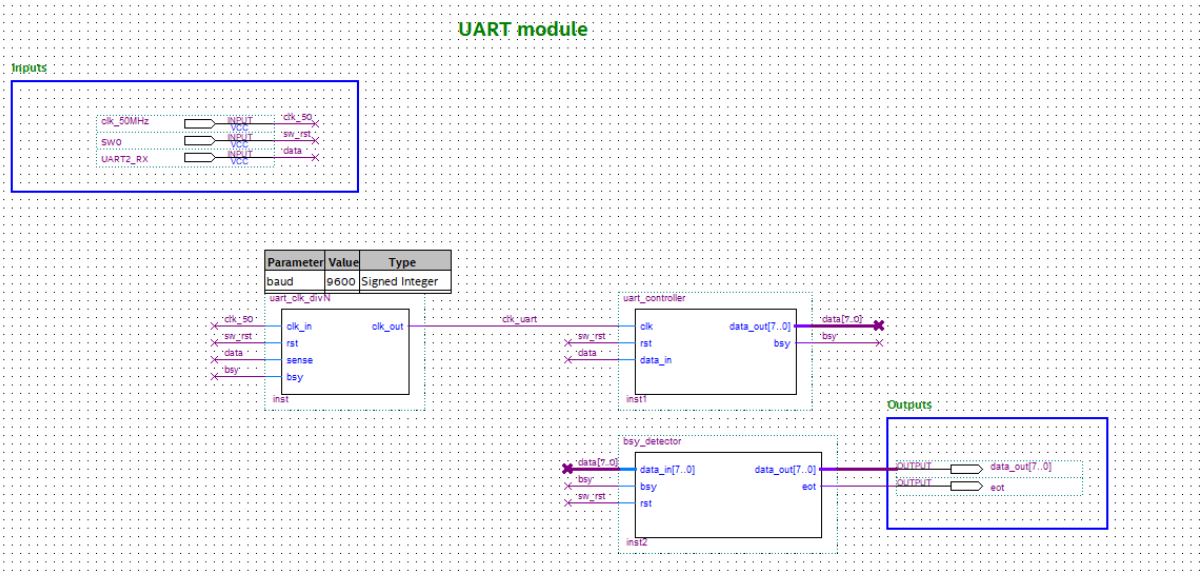


Figure 116 – the UART wired up. Parameterisable so support for any baud rate is included! *N.B. This only implements the receiver.*

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

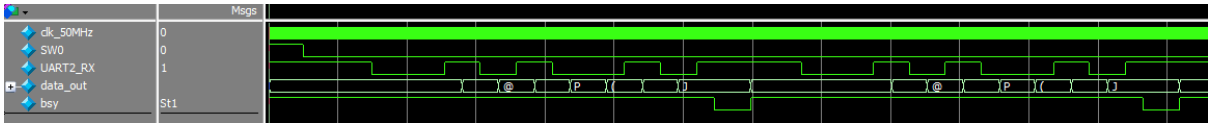


Figure 117 – UART testbench waveform. Data is not ready to be ready until the internal SPSR is done shifting its contents.

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.

2) 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.

```
always_latch begin

    if (rst) begin

        data_out <= 'd0;

    end

    // only read if bsy flag is low
    else if(!bsy) begin

        data_out <= data_in;
        eot <= 1;


    end

    else eot <= 0;


end
```

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 made 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)

1) 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.

```
module intermediaryRAM (

    output logic [7:0] data_out,
    output logic [7:0] opcode, sel, op1h, op1l, op2h, op2l,
    output logic rdy,

    input logic [7:0] data_in,
    input logic MODE,
    input logic clk, rst, DEBUG, load

);
```

Figure 119 – module IO ports.

```
// RAM creation
logic [7:0] ram [5:0];

// order is important!
assign opcode = ram[0];
assign sel = ram[1];
assign op1h = ram[2];
assign op1l = ram[3];
assign op2h = ram[4];
assign op2l = ram[5];
```

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, once 6 bytes were received.

```
// WRITE
0 : begin

    // clear other address registers upon switching modes
    addressDEBUG_reg <= 'd0;
    data_out <= 'd0;


    if (load) begin

        ram[addressWRITE_reg] <= data_in;
        addressWRITE_reg <= addressWRITE_reg + 1;

    end

    if(addressWRITE_reg == 'd6) begin

        addressWRITE_reg <= 'd0;
        rdy <= 1;

    end

    else rdy <= 0;  // do not want this to latch!


end
```

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.

```
if(load) begin

    ram[address_cnt] <= {opcode[3:0], sel[1:0], op1h, op1l, op2h, op2l};
    address_cnt <= address_cnt + 1;

end
```

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

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

```
if(!btnB) begin

    debugA <= ram[debug_address_cnt][37:34];
    debugB <= ram[debug_address_cnt][33:32];
    debugC <= ram[debug_address_cnt][31:24];

end

else begin

    debugA <= ram[debug_address_cnt][23:16];
    debugB <= ram[debug_address_cnt][15:8];
    debugC <= ram[debug_address_cnt][7:0];

end
```

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.

```
// idle state
2'd0 : begin

    if (d) begin
        state_reg <= 'd1;   // go to next state
    end


end

// pressed state
2'd1 : begin

    // wait for 100ms
    // can't use parameters in logical expressions
    // use unsigned int instead
    if (counter_reg != N) begin

        counter_reg <=  counter_reg + 1;

    end

    // then, do this
    else begin

        q <= 1;

        if (!d) begin
            state_reg <= 'd2;   // go to next state
            counter_reg <= 'd0;
        end

    end


end


// released state
2'd2 : begin

    // wait for 100ms
    if (counter_reg != N) begin

        counter_reg <=  counter_reg + 1;

    end

    // then, do this
    else begin

        q <= 0;
        counter_reg <= 'd0;
        state_reg <= 'd0;   //  loop back to idle state


    end


end
```

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.

```
current <= btnA;    // cycle 1
prev <= current;    // cycle 2


// rising edge detector:
// this acts as a "guard"
// without it (i.e. level triggered), since the circuitry updates 9600 times/sec,
// it will "see" many button inputs - bad!
// e.g. NOT if(btnA)...
if(current && !prev) begin

    if(debug_address_cnt != NUMBER_OF_INSTRUCTIONS-1) debug_address_cnt <= debug_address_cnt + 1;


end
```

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.*

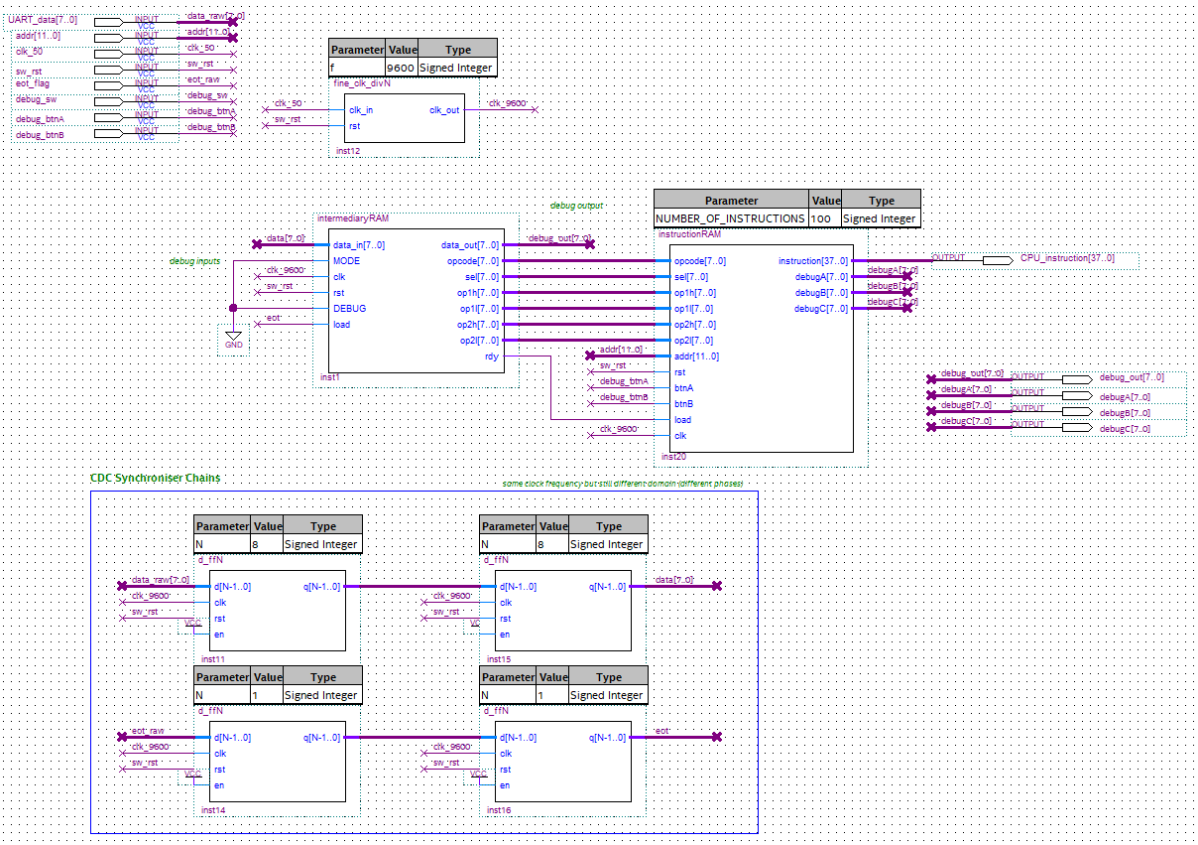


Figure 126 – *instructionBuffer* module fully wired.

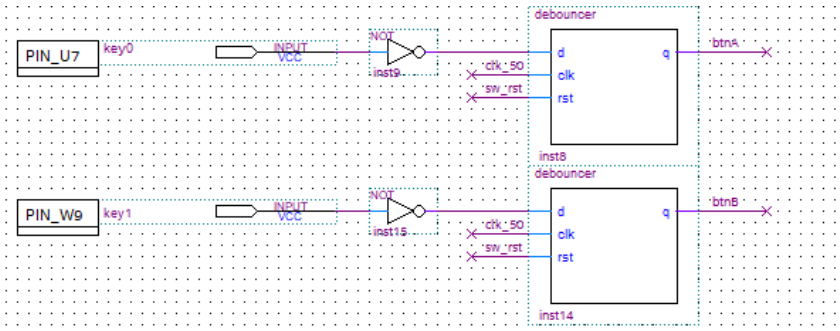


Figure 127 – *debouncer* modules in use for the debug buttons.

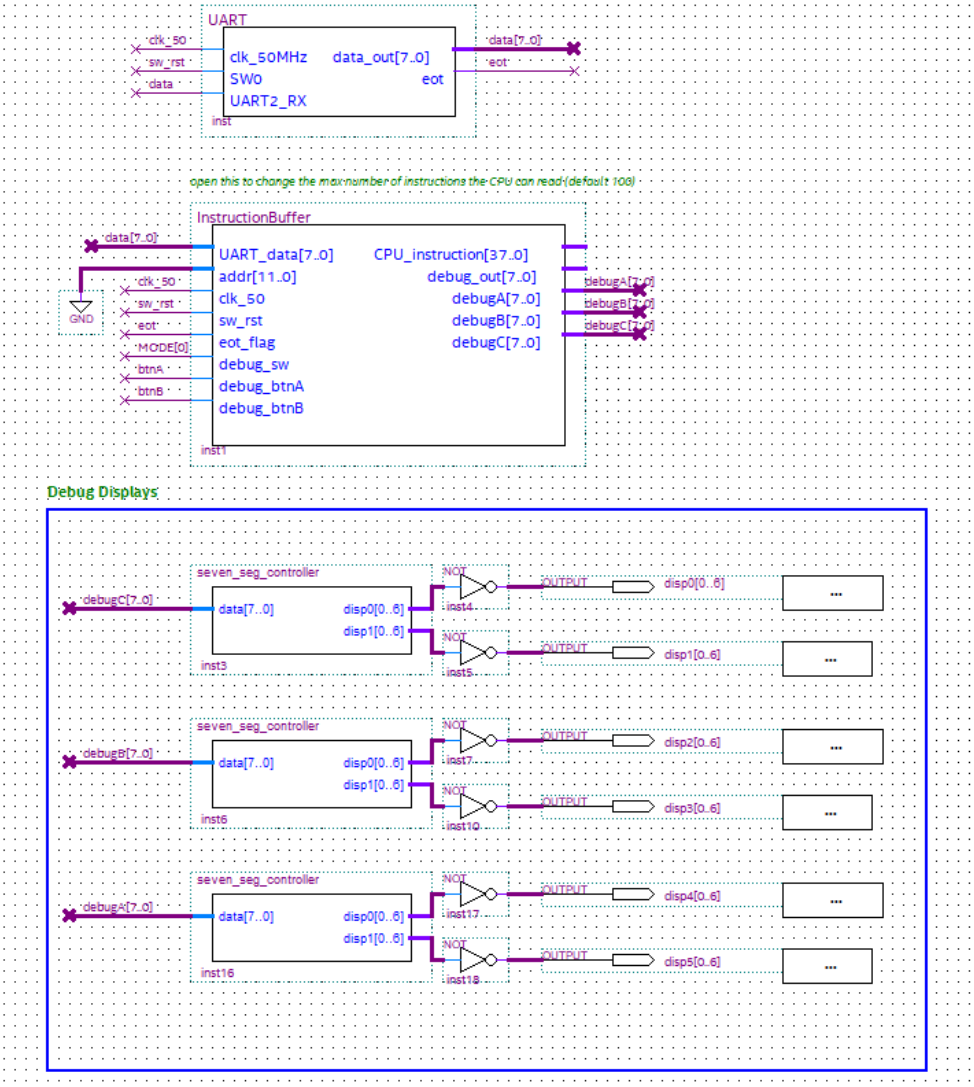


Figure 128 – UART and buffer connected. *(CPU address line was grounded for testing; CPU was not built at this stage).*

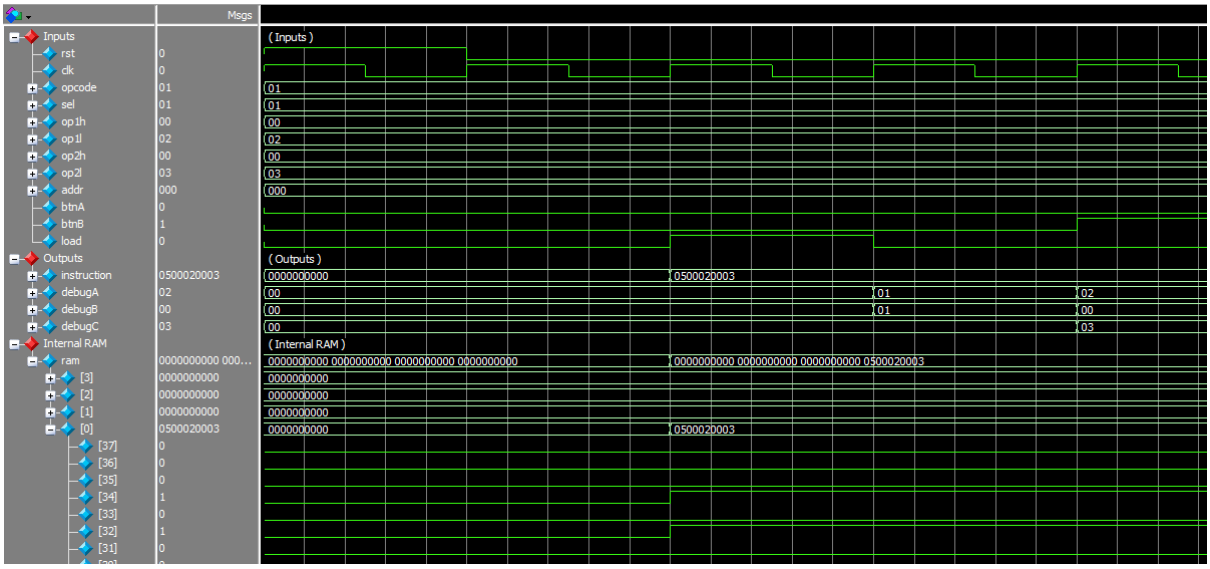


Figure 129 – *instructionRAM* testbench with a maximum of 4 instructions.

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

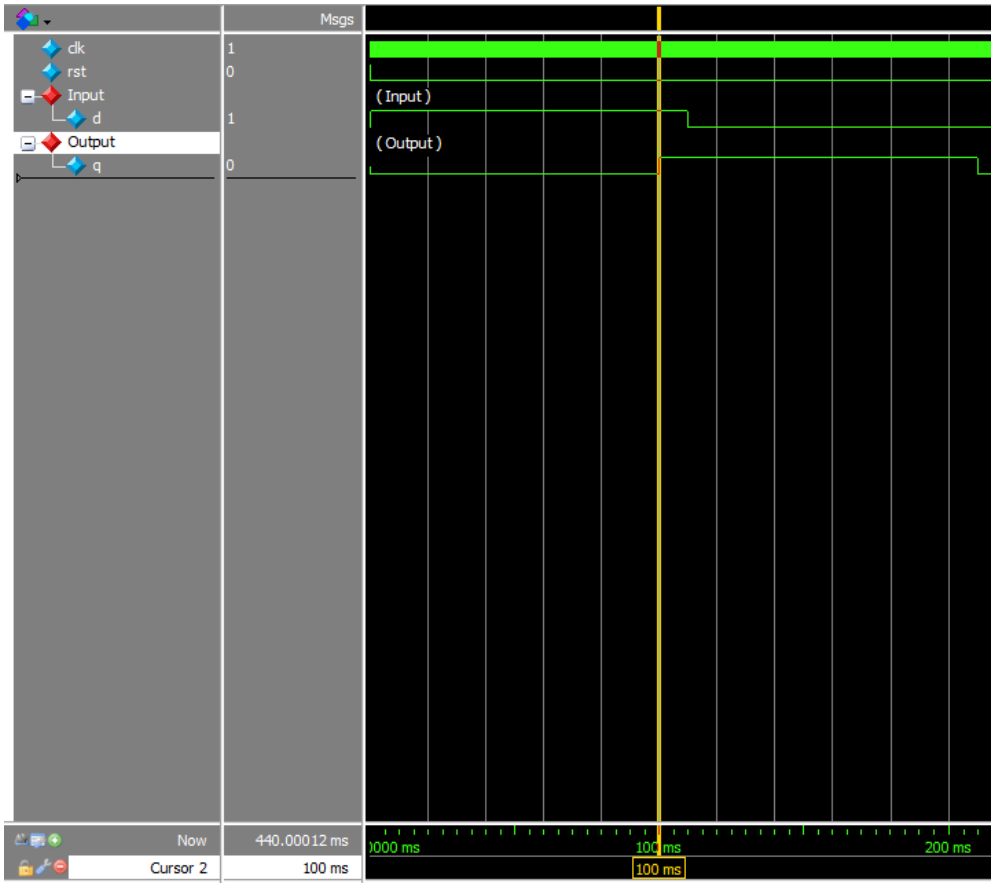


Figure 130 – *debouncer* testbench. Note how the output q goes high 100ms after the input d goes high.

Critical 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 for 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.

Flow Summary

<<Filter>>

Flow Status	Successful - Sat Apr 6 09:31:18 2024
Quartus Prime Version	23.1std.0 Build 991 11/28/2023 SC Lite Edition
Revision Name	uartt
Top-level Entity Name	TopLevel
Family	Cyclone V
Device	5CEBA4F23C7
Timing Models	Final
Logic utilization (in ALMs)	2,096 / 18,480 (11 %)
Total registers	4087
Total pins	53 / 224 (24 %)
Total virtual pins	0
Total block memory bits	0 / 3,153,920 (0 %)
Total DSP Blocks	0 / 66 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 4 (0 %)
Total DLLs	0 / 4 (0 %)

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.

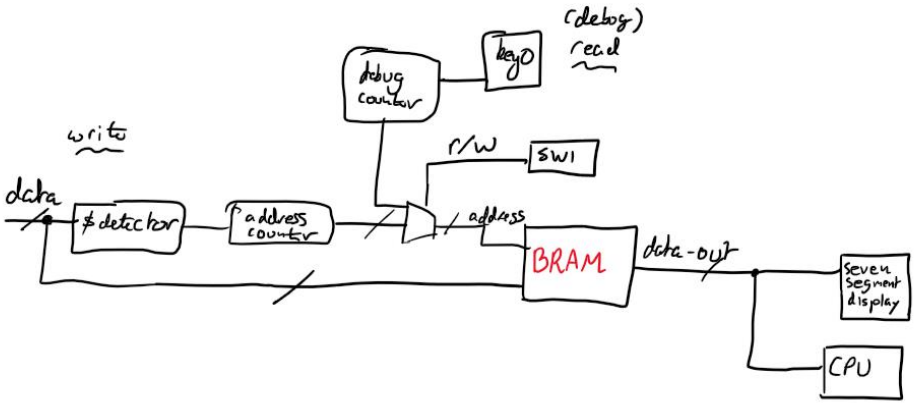


Figure 132 – example of how logic controlling the BRAM may be externalised.

Verilog RTL syntax example:

For example, in the following code, the `ramstyle` synthesis attribute specifies that the inferred RAM `my_ram` should be implemented using an :

```
(* ramstyle = "M144K" *) reg [0:7] my_ram[0:63];
```

Note: You can also embed the `ramstyle` synthesis attribute in a comment following the Variable Declaration of an inferred RAM, as shown in the following code:

```
reg [0:7] my_ram[0:63] /* synthesis ramstyle = "M144K" */;
```

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

Instead, IP blocks may be initialised and used:

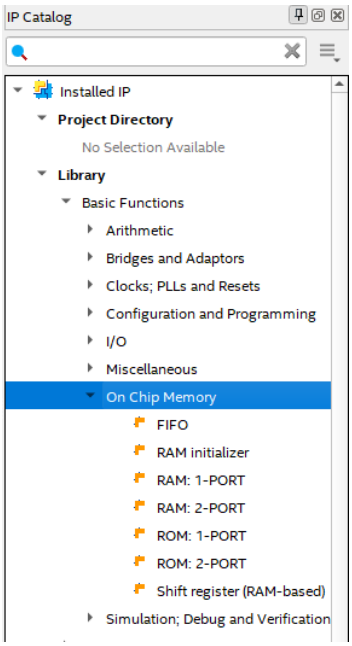


Figure 134 – Intel Quartus IP Catalog.

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:

		Member Code				
Resource		**A2**	**A4**	**A5**	**A7**	**A9**
Logic Elements (LE) (K)		25	49	77	150	301
ALM		9,430	18,480	29,080	56,480	113,560
Register		37,736	73,920	116,320	225,920	454,240
Memory (Kb)	M10K	1,760	3,080	4,460	6,860	12,200
	MLAB	196	303	424	836	1,717
Variable-precision DSP Block		25	66	150	156	342
18 x 18 Multiplier		50	132	300	312	684
PLL		4	4	6	7	8
GPIO		224	224	240	480	480
LVDS	Transmitter	56	56	60	120	120
	Receiver	56	56	60	120	120
Hard Memory Controller		1	1	2	2	2

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).*

```
// simulating an instruction
transmit(0x01); // opcode
transmit(0x01); // sel
transmit(0x00); // op1h
transmit(0x02); // op1l
transmit(0x00); // op2h
transmit(0x03); // op2l
```

Figure 136 – code to send the 6-byte instruction from the MCU to the FPGA.

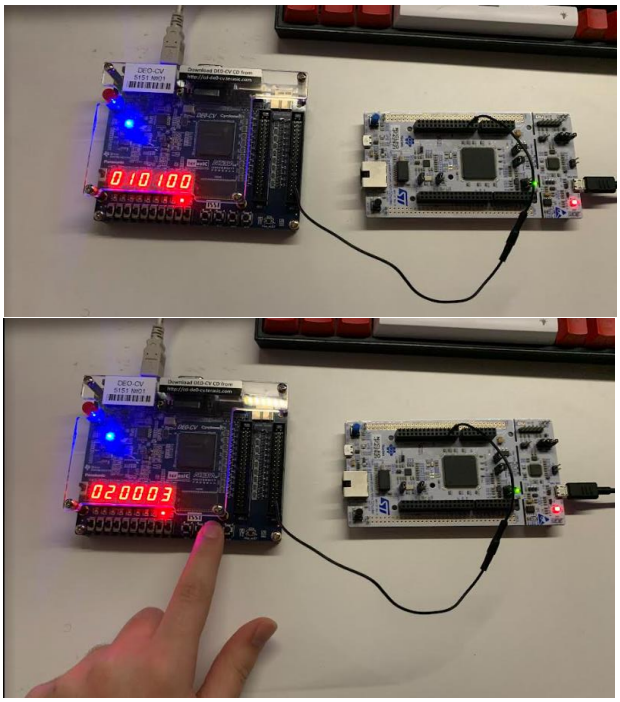


Figure 137 – viewing the received instruction. Top: top 3 bytes. Bottom: bottom 3 bytes. These results match the testbench waveform in Figure 129. *Note the single wire interface.*

8.5 The CPU

The first thing to implement was a generic, parameterisable CPU register component that can be used when needed. Its inputs would include an increment line as some registers would need that functionality.

```
case({load, inc})

2'b10 : begin

    internal_reg <= d;

end

2'b01 : begin

    internal_reg <= internal_reg + 1;

end

// else, (two) memory conditions -- valid!


endcase
```

Figure 138 – snippet of *CPU_regN* module code. This module would be used to for all the CPU's internal registers.

Next, system bus contention would have to be accounted for.

Critical Problem – Tri-state logic, Safety and Synthesisable RTL Code.

To prevent more than one register from placing its contents on the system bus at one go (bus contention), the use of tri-state buffers would traditionally be used. In RTL design, this buffer uses tri-state logic where there are 3 logic levels defined: 1) logic level low (0 or GND), 2) logic level high (1 or VCC) and 3) high impedance (Z or disconnected).

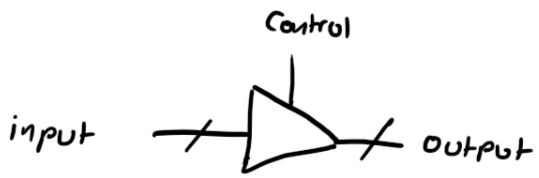


Figure 139 – active high tri-state buffer. If control is high, output = input; otherwise, output = Z.

The problem with this approach is that modern synthesis tools (e.g. Intel Quartus or AMD Vivado) do not synthesise tri-state buffers into the FPGA fabric. This is done for safety concerns due to potential short circuit issues if not managed properly. If tri-state RTL code is attempted to be synthesised, one of two things may happen. The compiler may give out errors and refuse to compile or, the design may be converted to "equivalent" MUX trees. The reason the word "equivalent" is placed in quotation marks is because the user cannot guarantee the MUX trees' behaviour is completely identical to the initial desired functionality of the modules that use tri-state logic – the synthesis tools are not perfect. In fact, what makes this even more problematic is the fact that simulation tools like ModelSim will not give the user any errors as they only check for syntax, not whether the RTL code is synthesisable on the target device or not.

So, as an alternative to using tri-state logic, a MUX (the SB mux) will be used instead. A multiplexer can achieve identical desired behaviour as it can only allow one output at a time. Recall that the registers that can access the system bus are:

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

So naturally, the SB mux will have 10 inputs, 1 output and a 4-bit select line (SYSBUSSEL).

```
module systemBus_muxN #(parameter N=38) (

    output logic [N-1:0] q,

    input logic [N-1:0] PC,DR,AR,AC,MEM,TR,rop1,rop2,GPR1,
    input logic [3:0] sel


);
```

Figure 140 – SB mux port definitions.

```
always_comb begin
    case(sel)

    4'd0 : begin
        q = PC;
    end

    4'd1 : begin
        q = DR;
    end

    4'd2 : begin
        q = AR;
    end

    4'd3 : begin
        q = AC;
    end

    4'd4 : begin
        q = MEM;
    end

    4'd5 : begin
        q = TR;
    end

    4'd6 : begin
        q = rop1;
    end

    4'd7 : begin
        q = rop2;
    end

    4'd8 : begin
        q = GPR1;
    end

    default : begin
        q = 'd0;    // does not matter what the output is here
    end
    endcase
end
```

Figure 141 – SB mux implementation code.

Next, the *flags_setter* module was implemented. This is the module that was responsible for setting the NZCV flags. There are only three instructions in the ISA that can set flags: add, sub and cmp.

```
module flags_setter (

    output logic N,Z,C,V,

    input logic [3:0] opcode,
    input logic [11:0] op1,op2,
    input logic [11:0] AC_result,   // concerned with the bottom 12 bits only
    input logic rst, ACLOAD

);
```

Figure 142 – *flags_setter* port definitions.

Note that the hardware for this module only updates when the ACLOAD signal is set.

```
// add
4'b1001 : begin

    instruction <= add;

    // N
    if(AC_result[11] == 1) begin
        N <= 1;
    end
    else begin
        N <= 0;
    end

    // Z
    if(AC_result == 'd0) begin
        Z <= 1;
    end
    else begin
        Z <= 0;
    end

    // C - set due to overflow
    // when result is smaller than either operand
    if(AC_result < op1 || AC_result < op2) begin
        C <= 1;
    end
    else begin
        C <= 0;
    end

    // V - sign bit changes to 1 when adding two positive numbers
    if(AC_result[11] == 1 && op1[11] == 0 && op2[11] == 0) begin
        V <= 1;
    end
    else begin
        V <= 0;
    end

end
```

Figure 143 – flag semantics for the add instruction.

```
// sub
4'b1010 : begin

    instruction <= sub;

    // N
    if(AC_result[11] == 1) begin
        N <= 1;
    end
    else begin
        N <= 0;
    end

    // Z
    if(AC_result == 'd0) begin
        Z <= 1;
    end
    else begin
        Z <= 0;
    end

    // C - set when borrow did NOT occur (op1 > op2)
    // so when result is smaller than either operand
    if(AC_result < op1 || AC_result < op2) begin
        C <= 1;
    end
    else begin
        C <= 0;
    end

    // V - sign bit changes to 0 when subtracting a positive number from a negative number
    if(AC_result[11] == 0 && op1[11] == 1 && op2[11] == 0) begin
        V <= 1;
    end
    else begin
        V <= 0;
    end


end
```

Figure 144 – flag semantics for the sub instruction. The cmp instruction was implemented in the same way as the sub instruction. This is because the cmp instruction is just a special form of the sub instruction (see Figure 39).

To implement GPR functionality to the CPU, the *GPR Selector* module was then constructed. To begin, the 32-bit data to be stored in one of the GPRs along with the GPRLOAD control signal had to be concatenated.

```
module conc32 (

    output logic [32:0] data_out,

    input logic [31:0] data_in,
    input logic LOAD
);

// a simple concatenator
// to be connected to selectors

assign data_out = {data_in,LOAD};


endmodule
```

Figure 145 – full concatenator module code.

Then, the GPR DEMUX and MUX where built.

```
module GPR_DEMUX (

    output logic [32:0] rA,rB,rC,rD,rE,rF,rG,rH,rI,rJ,rK,rL,rM,  // packed arrays causes problems when wiring :(

    input logic [32:0] data_in,
    input logic [3:0] sel

);

always_comb begin

    case(sel)

    // each case must address ALL outputs to infer purely comb logic
    // otherwise - latch!
    4'd0 : begin
        rA = data_in;
        {rB,rC,rD,rE,rF,rG,rH,rI,rJ,rK,rL,rM} = 'd0;
    end

    4'd1 : begin
        rB = data_in;
        {rA,rC,rD,rE,rF,rG,rH,rI,rJ,rK,rL,rM} = 'd0;
    end

module GPR_MUX (

    output logic [31:0] data_out,

    input logic [31:0] rA,rB,rC,rD,rE,rF,rG,rH,rI,rJ,rK,rL,rM,
    input logic [3:0] sel

);

always_comb begin

    case(sel)

    4'd0 : begin
        data_out = rA;

    end

    4'd1 : begin
        data_out = rB;

    end
```

Figure 146 – snippet of GPR DEMUX and MUX codes.

Problems:

- Despite the GPRs being named (and referred to in this text) as r0-r12, using numbers in the names of ports causes naming conflicts in Quartus. Hence, letters were used instead to circumvent this (rA and rB etc.).
- A minor oversight during the design of this module was made. The outputs of any two GPRs, GPROUT1 and GPROUT2 were selected by rop1[15..12] and rop2[15..12]. This does not account for every single AL instruction (except for mvn) which use rop1[11..8] and rop2[11..8] as their select bits. To account for this, a module had to be built that uses these 4 bits instead when the CPU fetches an AL instruction.

```
module dest_reg_selector (

    output logic [3:0] GPR_sel1,
    output logic [3:0] GPR_sel2,

    input logic [39:0] state,
    input logic [3:0] rop1,     // 15..12 (default)
    input logic [3:0] rop2,     //
    input logic [3:0] rop1_AL,  // 11..8
    input logic [3:0] rop2_AL   // 11..8

);

always_comb begin

    // add1, sub1, mul1, lsr1, add1, or1
    // 2**n = 1<<n
    if(state == 2**26 || state == 2**28 || state == 2**30 || state == 2**32 || state == 2**34 || state == 2**36) begin
        GPR_sel1 = rop1_AL;
        GPR_sel2 = rop2_AL;
    end

    else begin
        GPR_sel1 = rop1;
        GPR_sel2 = rop2;
    end



end


endmodule
```

Figure 147 – the module that fixes the second problem mentioned above. Where “AL” stands for Arithmetic and Logic instruction.

The wiring of the CPU thus far starts on the next page.

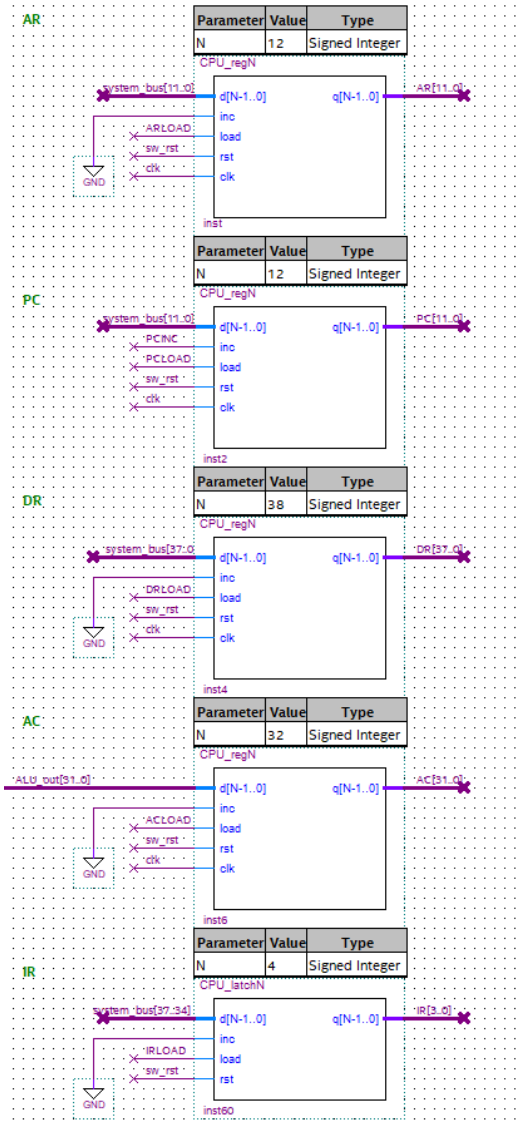


Figure 148 – the CPU's fundamental registers.

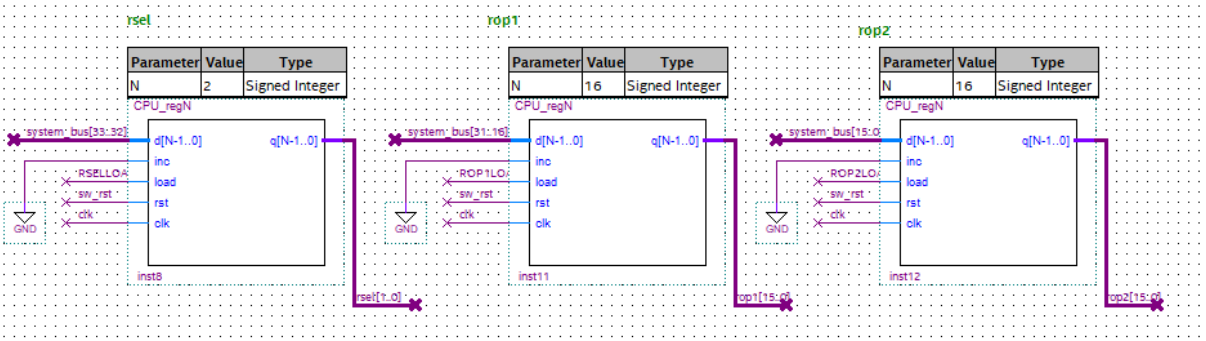


Figure 149 – the three registers that hold the 2 select bits, and the 2x16-bit operands.

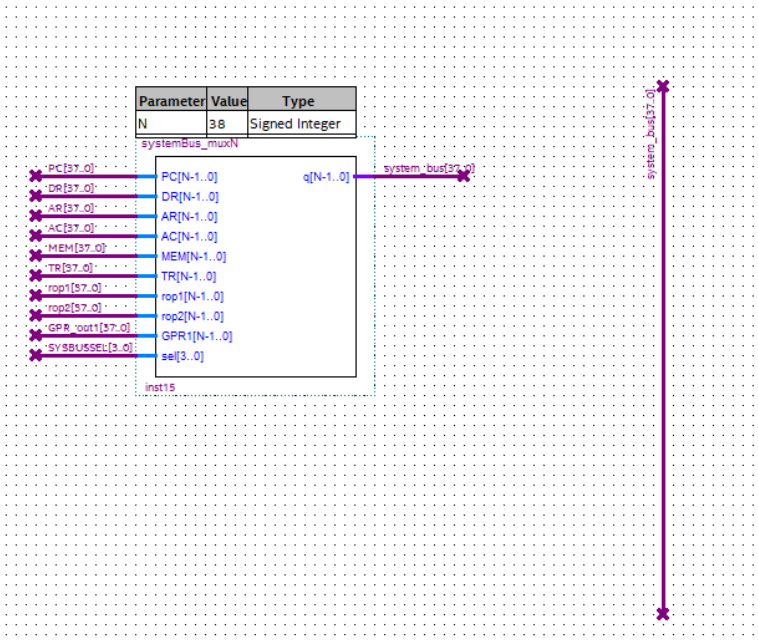


Figure 150 – the SB mux along with the 38-bit system bus.

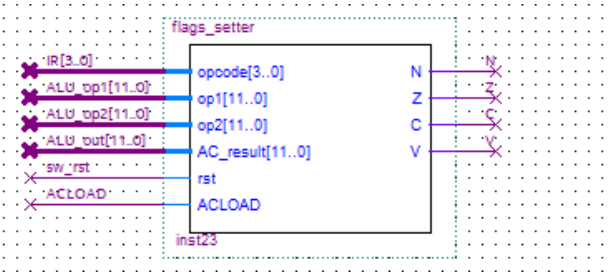


Figure 151 – the *flags_setter* module.

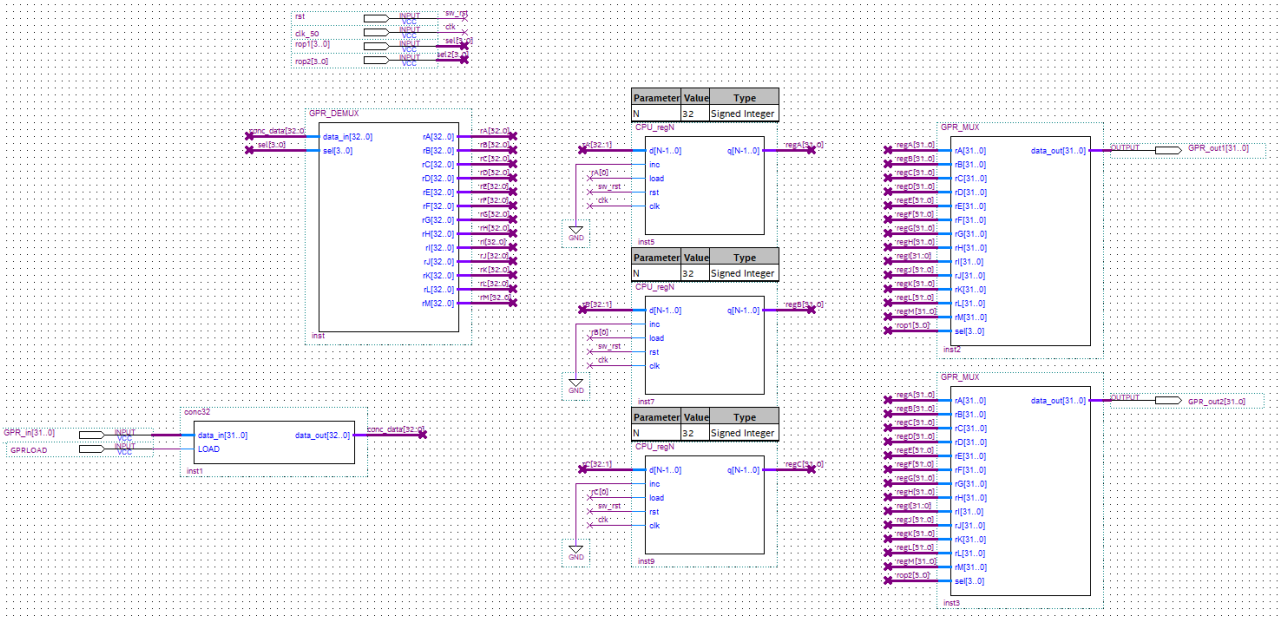


Figure 152 – the *GPR selector* module with r0, r1 and r2 shown.

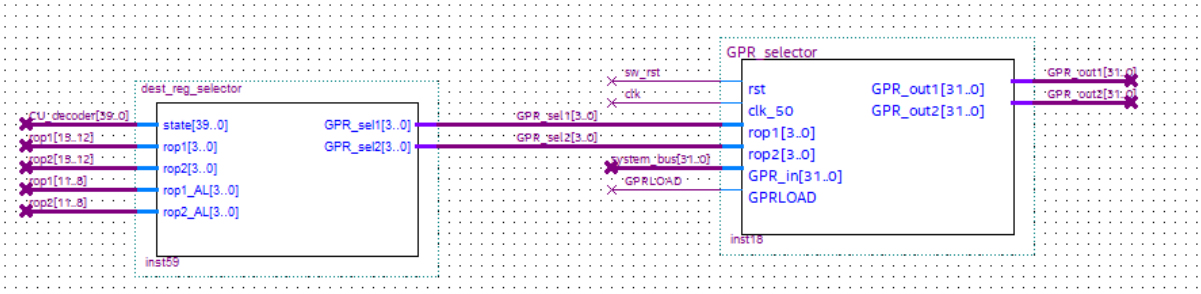


Figure 153 – the *GPR selector* module with the *dest_reg_selector* module that addresses problem 2 above.

The design of the last two remaining components of the CPU, the ALU and CU will now be discussed.

The simpler of the two, the ALU, was implemented next, with an internal MUX that chooses which result to output. Its output is hard-wired to the input of the AC. This means that the ALU will always assert something on its output line. This however was not an issue as the AC's output will only update when the ACLOAD is asserted by the CU.

```
case (ALUSEL)

3'd0 : begin
    q = op1 + op2;
end

3'd1 : begin
    q = op1 - op2;
end

3'd2 : begin
    q = op1 * op2;
end

3'd3 : begin
    q = op1 >> op2;
end

3'd4 : begin
    q = op1 & op2;
end

3'd5 : begin
    q = op1 | op2;
end

3'd6 : begin
    q = ~op2;
end


// needed otherwise will not be realisable with comb logic!
default : begin
    q = op1 + op2;  // addition by default (does not really matter what goes here)
end
```

Figure 154 – the ALU code. op1 and op2 are its two operand inputs.

To choose the two operands, two MUXs named *operand_MUX* would be used to select between the 12-bit literal operand found in registers rop1[11..0] and rop2[11..0], or two GPRs (the outputs of the *GPR Selector* module; GPROUT1 and GPROUT2).

```
module operand_MUX (

    output logic [31:0] data_out,

    input logic [11:0] rop,
    input logic [31:0] GPR,
    input logic sel

);

always_comb begin

    if(!sel) begin
        data_out = {20'd0,rop}; // clears the top 20 bits just to be safe
    end

    else begin
        data_out = GPR;
    end


end


endmodule
```

Figure 155 – the *operand_MUX* code.

With the ALU circuitry now complete, the wiring is shown below.

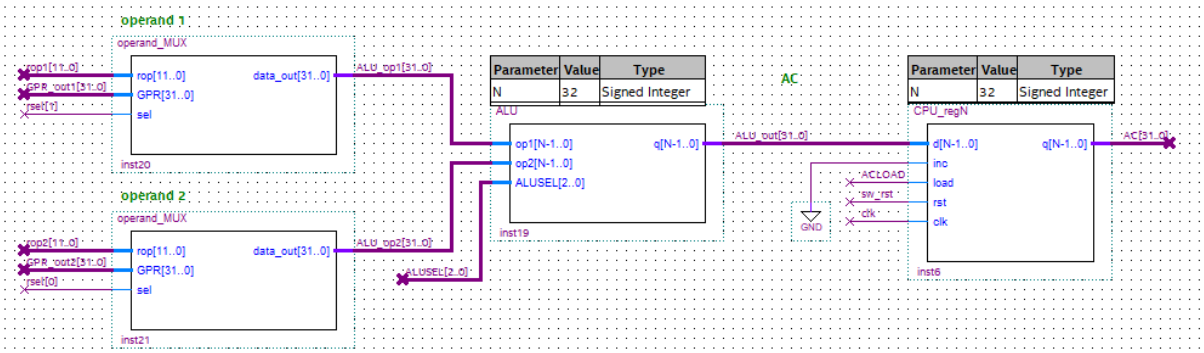


Figure 155 – the complete ALU circuity. *Note the use of the select bits here.*

Finally, the Control Unit was designed. The three parts of the CU were implemented in the following order.

1. The State Counter

The up-counter responsible for taking in the opcode from the IR and outputting a value corresponding to the start state of the instruction. This is the module that had to account for conditional state execution based on the select bits (e.g. a choice between mov or ALTmov). This is shown below.

```
module CU_counter #(parameter opcode_bits=4, N=6) (

    output logic [N-1:0] q,

    input logic [opcode_bits-1:0] opcode,   // connected to IR
    input logic [1:0] sel_bits,             // connected to rsel
    input logic load, inc, clr, clk, rst


);
```

Figure 156 – *CU_counter* port definitions.

```
// mov
4'd1 : begin
    if(!select_bits[0]) begin
        counter_reg <= 'd4;
    end

    // ALTmov
    else begin
        counter_reg <= 'd5;
    end
end

// ldr
4'd2 : begin
    if(!select_bits[0]) begin
        counter_reg <= 'd7;
    end

    // ALTload
    else begin
        counter_reg <= 'd9;
    end
end

// str
4'd3 : begin

    if(!select_bits[0]) begin
        counter_reg <= 'd13;
    end

    // ALTstr
    else begin
        counter_reg <= 'd17;
    end
end
```

Figure 157 – using select bits to choose correct execution state. mov, ldr and str are shown above. *See Figure 59 for the counter mapping.*

2. The State Decoder

```
module CU_decoder #(parameter N=6, states=40) (

    output logic [states-1:0] CPU_state,
    input logic [N-1:0] counter_value

);

// loop var
int unsigned i=0;

// always_comb:  does not infer purely combinational logic for some reason...
// so a nice trick: use always_latch :winky-face:
always_latch begin

    // 2**N of counter_value to not need a "default" statement
    // and hence be synthesisable by comb logic and a for loop
    // as we cover all input combinations for "counter_value"
    for(i=0;i<2**N;i++) begin

        case (counter_value)

        i : begin
            CPU_state = 1<<i;
        end

        endcase

    end


end
```

Figure 158 – *CU_decoder* implementation. Note the use of always_latch and for loops here.

3. The Control Signal Logic Controller

The largest and most complex part of the CU. But it is also the final component.

```
module CU_logic #(parameter states=40) (


    output logic RSELLOAD,ROP1LOAD,ROP2LOAD,TRLOAD,ARLOAD,PCLOAD,DRLOAD,ACLOAD,IRLOAD,GPRLOAD,MEMLOAD,
    PCINC,
    COUNTER_LD,COUNTER_INC,COUNTER_CLR,

    output logic [2:0] ALUSEL,

    output logic [3:0] SYSTEMBUSSEL,

    input logic [states-1:0] CPU_state,

    input logic N,Z,C,V

);
```

Figure 159 – port definitions. Note how the outputs are all the control signals the CPU needs to control.

```
case(CPU_state)

// fetch1
40'd2**0 : begin

    {COUNTER_INC,ARLOAD} = 2'b11;
    {RSELLOAD,ROP1LOAD,ROP2LOAD,TRLOAD,PCLOAD,DRLOAD,ACLOAD,IRLOAD,GPRLOAD,MEMLOAD,PCINC,COUNTER_LD,COUNTER_CLR} = 'd0;
    ALUSEL = 'd0;
    SYSTEMBUSSEL = 'd0;

    CPUstate = fetch1;
end

// fetch2
40'd2**1 : begin

    {COUNTER_INC,DRLOAD,PCINC} = 3'b111;
    {RSELLOAD,ROP1LOAD,ROP2LOAD,TRLOAD,ARLOAD,PCLOAD,ACLOAD,IRLOAD,GPRLOAD,MEMLOAD,COUNTER_LD,COUNTER_CLR} = 'd0;
    ALUSEL = 'd0;
    SYSTEMBUSSEL = 'd4;

    CPUstate = fetch2;
end

// fetch3
40'd2**2 : begin

    {COUNTER_LD,RSELLOAD,ROP1LOAD,ROP2LOAD,IRLOAD} = 5'b11111;
    {TRLOAD,ARLOAD,PCLOAD,DRLOAD,ACLOAD,GPRLOAD,MEMLOAD,PCINC,COUNTER_INC,COUNTER_CLR} = 'd0;
    ALUSEL = 'd0;
    SYSTEMBUSSEL = 'd1;

    CPUstate = fetch3;
end
```

Figure 160 – implementation of the fetch1, fetch2 and fetch3 execution routines.

When compared with the ISA control signals that was derived, the above code becomes more digestible.

```
Counter lines

LD  = fetch3
INC = fetch1 | fetch2 | ALTmov1 | ldr1 | ALTldr1 | ALTldr2 | ALTldr3 | str1 | str2 | str3 | ALTstr1 | ALTstr2 | ALTstr3 | add1 | sub1 | mul1 | lsr1 | and1 | or1 | mvn1
CLR = nop1 | mov1 | ALTmov2 | ldr2 | ALTldr4 | str4 | ALTstr4 | cmp1 | b1 | bgt1 | blt1 | beq1 | add2 | sub2 | mul2 | lsr2 | and2 | or2 | mvn2


Load lines (LHS)

RSELLOAD = fetch3
ROP1LOAD = fetch3
ROP2LOAD = fetch3
TRLOAD   = ALTmov1 | str2 | ALTstr2
ARLOAD   = fetch1 | ldr1 | ALTldr1 | str1 | ALTstr1
PCLOAD   = b1 | bgt1 | blt1 | beq1
DRLOAD   = fetch2 | ALTldr3 | str3 | ALTstr3
ACLOAD   = cmp1 | add1 | sub1 | mul1 | lsr1 | and1 | or1 | mvn1
IRLOAD   = fetch3
GPRLOAD  = mov1 | ALTmov2 | ldr2 | ALTldr2 | ALTldr4 | add2 | sub2 | mul2 | lsr2 | and2 | or2 | mvn2


Increment lines

PCINC = fetch2


Memory load line

MEMLOAD = str4 | ALTstr4


System bus access (RHS) (number is the SYSTEMBUSSEL value)

PC   = fetch1                                                                0
DR   = fetch3 | ALTldr4 | str4 | ALTstr4                                     1
AR   = ldr2 | ALTldr2                                                        2
AC   = add2 | sub2 | mul2 | lsr2 | and2 | or2 | mvn2                         3
MEM  = fetch2 | ALTldr3                                                      4
TR   = ALTmov2 | str3 | ALTstr3                                              5
rop1 = ALTstr2                                                               6
rop2 = mov1 | ldr1 | ALTldr1 | str1 | ALTstr1 | b1 | bgt1 | blt1 | beq1     7
GPR1 = ALTmov1 | str2                                                        8
```

Figure 161 – state signals that need to be asserted during state fetch1 are highlighted above.

Finally, the complete Control Unit wiring is shown below.

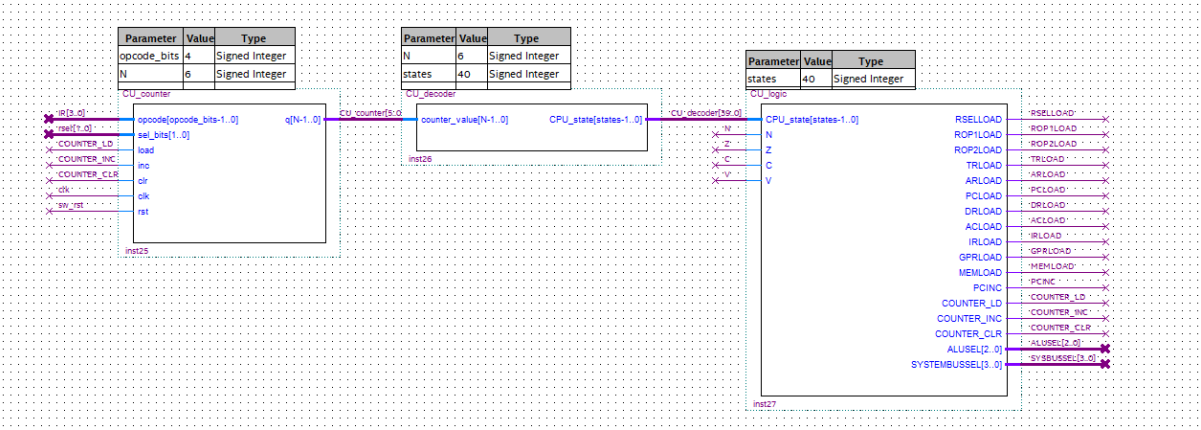


Figure 162 – The complete Control Unit.

The testing of the full computer (CPU + memory) will be in Section 9.

8.6 Memory

To complete the computer, the CPU needs memory to read from. The only thing notable to mention here is the *address checker* module. This module implements the memory map found in Figure 73.

```
always_comb begin

    // Instructions
    if (addr >= 12'h000 && addr <= 12'h400) begin
        sel = 'd0;
        {load_mmr,load_stack} = 2'b00;
        state = instructions;
    end

    // Memory mapped registers
    else if (addr >= 12'h401 && addr <= 12'h44B) begin
        sel = 'd1;
        {load_mmr,load_stack} = {MEMLOAD,1'b0};
        state = mmr;
    end

    // Stack memory
    else if (addr >= 12'h44C && addr <= 12'hFFF) begin
        sel = 'd2;
        {load_mmr,load_stack} = {1'b0,MEMLOAD};
        state = stack;
    end

    else begin
        sel = 'd0;
        {load_mmr,load_stack} = 2'b00;
        state = instructions;
    end


end
```

Figure 163 – implementation of the memory map. Note how there is no load signal for the instruction memory (read only).

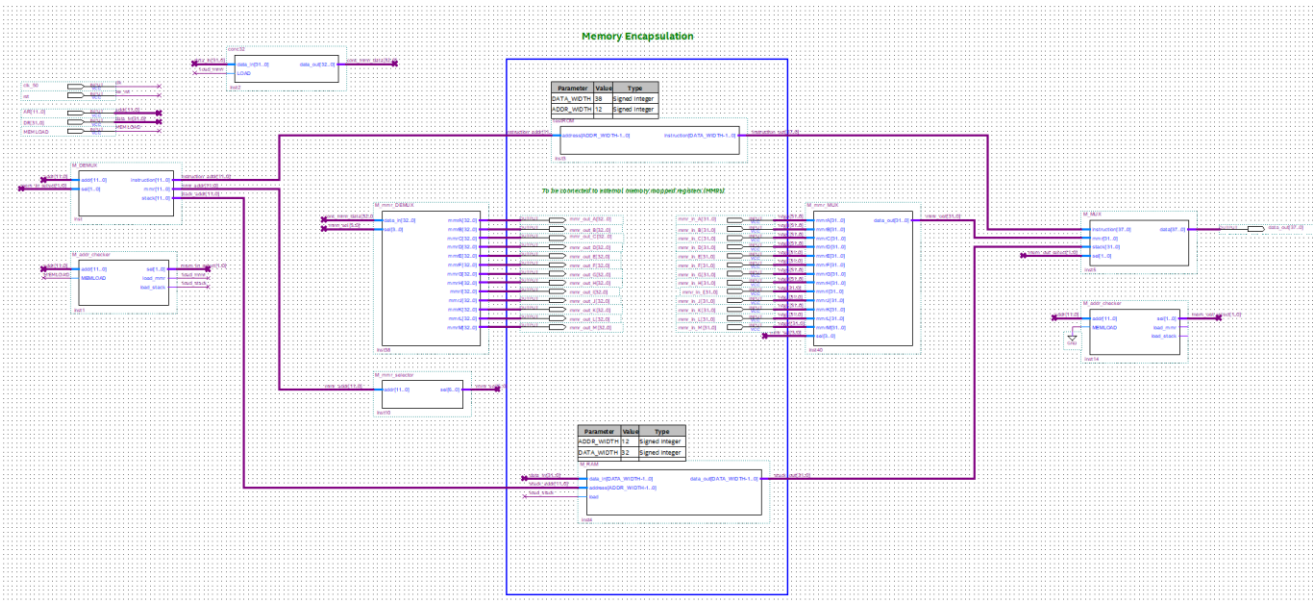


Figure 164 – The three subtypes of memory in the blue box. Top: (test) instruction ROM. Middle: MMRs. Bottom: stack memory.

Two notable things to mention:

- For testing purposes, the instruction ROM was temporarily replaced with a test instruction ROM. Its contents will be explained in the next section.
- The memory mapped registers are external to the memory module. This is what a memory mapped register is: an external peripheral control register that the CPU can access using the same address bus (hence the same address space as the instruction and stack memories).

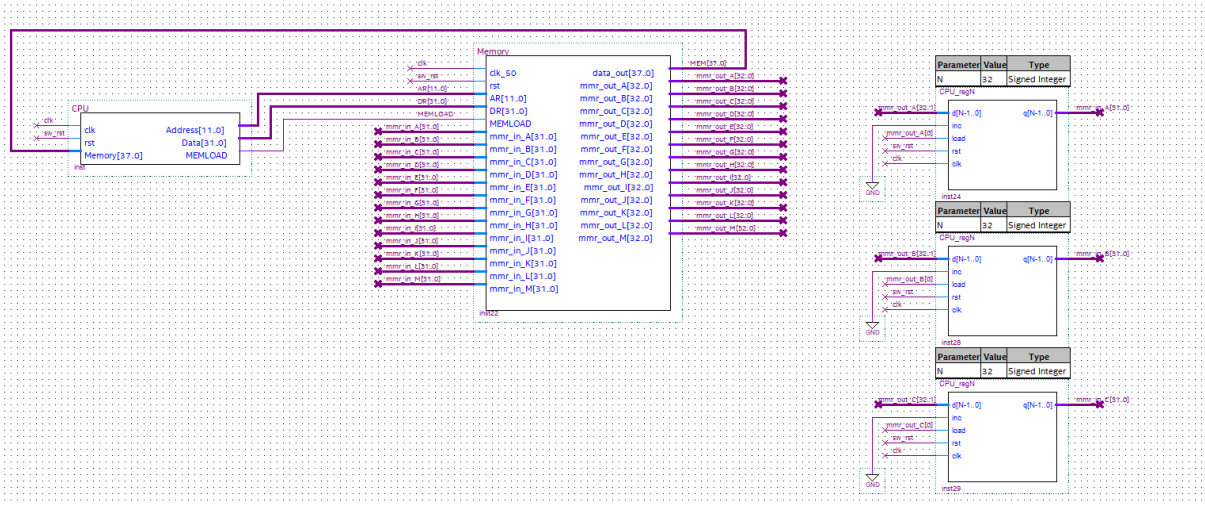


Figure 165 – connecting the CPU, memory and MMRs (three are shown here).

8.7 The LED Panel

This LED panel revolved around using WS2812B addressable LEDs. So, a controller that implements its protocol would be required. Since the idea of this came very late into the development cycle, the research of this protocol had to be done at this stage. All information regarding this protocol is from [21] and [22].

8.7.1 The WS2812B Protocol

This protocol uses a single data line to send data to N addressable LEDs. Each LED requires 24 bits of colour information, 8 bits for each colour: green, red, and blue (GRB), in that order.

Composition of 24bit data:

G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	B7	B6	B5	B4	B3	B2	B1	B0

Note: Follow the order of GRB to sent data and the high bit sent at first.

Figure 166 – 24-bit data frame; the MSB must be sent first.

Each one of those bits may be either a 1 or a 0. In this protocol, a 1 is represented by what is called a “1 code” and a 0 is represented by a “0 code”.

The 0 code: *consists of a logic level high for* ***400ns ± 150ns****, followed by a logic level low for* ***850ns ± 150ns****.*

The 1 code: *consists of a logic level high for* ***800ns ± 150ns****, followed by a logic level low for* ***450ns ± 150ns****.*

After the colour data is sent to all N LEDs (so after sending 24*N bits), a “reset code” needs to be sent to tell the LEDs that transmission is over, and to latch and light up.

The reset code: *data line to be held low for at least* ***50us****.*

The codes are shown graphically on the next page.

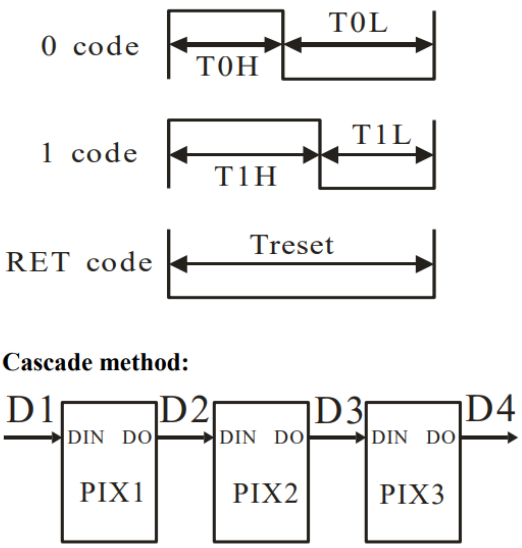


Figure 167 – data frame structure. Also shown is an example of how three LEDs would be cascaded.

Problem: there were two variants of this addressable LED: the *WS2812B* as well as the *WS28128B-2020*. Unfortunately, it is not usually apparent to the buyer which LED type will be received upon buying the strip. The *-2020* version of the LED differs in code timings. However, when comparing the high and low codes for both LED types, it is possible to choose timings that would work regardless of which LED the user has. More distinctly, the reset code of the *-2020* version requires at least **280us** as opposed to 50us. This can very easily be circumvented by using a 300us reset code time – would work with both LED types.

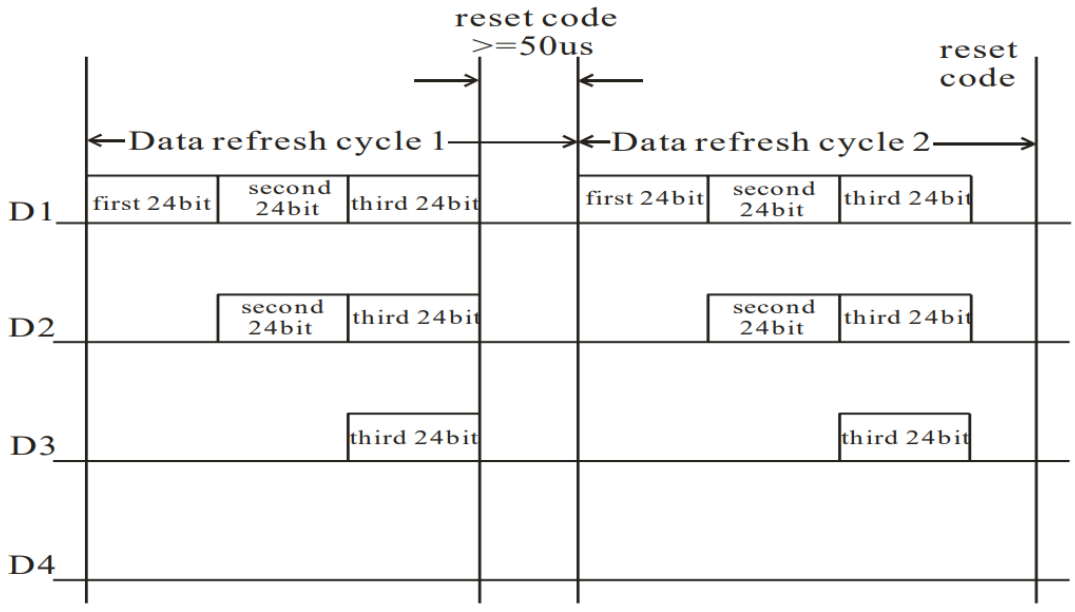


Figure 168 – sending data to 3 cascaded LEDs (WS28128B).

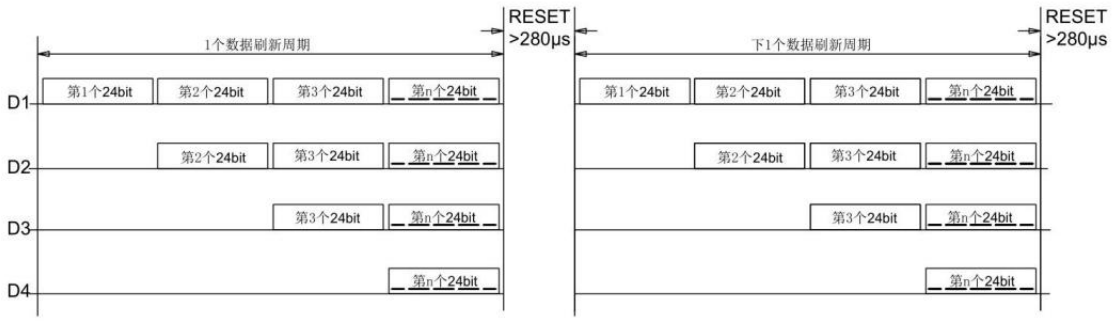


Figure 169 – sending data to 4 cascaded LEDs (WS28128B-2020).

Data transfer time(TH+TL=1.25μs±600ns)

T0H	0 code ,high voltage time	0.4us	±150ns
T1H	1 code ,high voltage time	0.8us	±150ns
T0L	0 code , low voltage time	0.85us	±150ns
T1L	1 code ,low voltage time	0.45us	±150ns
RES	low voltage time	Above 50μs	

Figure 170 – WS2812B code times.

Data Transfer Time

T0H	0 code, high voltage time	220ns~380ns
T1H	1 code, high voltage time	580ns~1μs
T0L	0 code, low voltage time	580ns~1μs
T1L	1 code, low voltage time	220ns~420ns
RES	Frame unit, low voltage time	>280μs

Figure 171 – WS2812B-2020 code times.

Example: suppose there are 3 LEDs cascaded together: D1, D2 and D3. The user wishes to light up D1 and D2 green and have D3 switched off. So, the GRB data needed is:

- **D1:** FF 00 00
- **D2:** FF 00 00
- **D3:** 00 00 00

And the codes sent will be:

- **D1:** 8 high codes; 8 low codes; 8 low codes
- **D2:** 8 high codes; 8 low codes; 8 low codes
- **D3:** 8 low codes; 8 low codes; 8 low codes

8.7.2 RTL Controller Implementation

This module would be parameterisable to allow control of N LEDs, along with a choice of colour. Hardware timers will be used to meet the protocol timing requirements. The controller will be connected to the FPGA's 50 MHz clock.

```
module WS2812B_controller #(parameter N=32, GRB_colour = 24'hFF0000) (

    output logic data_out,

    input logic [N-1:0] data_in,
    input logic clk_50, rst

);
```

Figure 172 – controller port definitions. Note the parameters for number of LEDs and desired colour. The drawback of this method is that every LED strip's colour will be static (a single colour). However, this suffices for the project.

```
// time taken to reach value, T = value/fclk
// thus value = T*fclk = T*50M
parameter N_300ns = 15;
parameter N_800ns = 40;
parameter N_500us = 25000;  // reset code time
```

Figure 173 – these times were chosen to ensure protocol timing is met regardless of LED type.

```
// low code
if(!GRB_reg[counterGRB_reg]) begin

    // T0H = 0.3us
    if(counter1_reg != N_300ns && counter2_reg == 'd0) begin

        counter2_reg <= 'd0; // reset other timer

        data_out <= 1;
        counter1_reg <= counter1_reg + 1;

    end

    // when first timer completed...
    // T0L = 0.8us
    else if(counter2_reg != N_800ns) begin

        counter1_reg <= 'd0; // reset other timer

        data_out <= 0;
        counter2_reg <= counter2_reg + 1;

    end

    // whens second timer completed...
    else begin

        counter2_reg <= 'd0; // reset other timer
        counterGRB_reg <= counterGRB_reg - 1;

    end


end
```

Figure 174 – low code implementation for when the GRB bit is low (T0H then T0L).

```
// high code
else begin

    // T1H = 0.8us
    if(counter1_reg != N_800ns && counter2_reg == 'd0) begin

        counter2_reg <= 'd0; // reset other timer

        data_out <= 1;
        counter1_reg <= counter1_reg + 1;

    end

    // when first timer completed...
    // T1L = 0.3us
    else if(counter2_reg != N_300ns) begin

        counter1_reg <= 'd0; // reset other timer

        data_out <= 0;
        counter2_reg <= counter2_reg + 1;

    end

    // whens second timer completed...
    else begin

        counter2_reg <= 'd0; // reset other timer
        counterGRB_reg <= counterGRB_reg - 1;

    end


end
```

Figure 175 – high code implementation for when the GRB bit is high (T1H then T1L).

Note that the above codes are iterated a total of 24*N times. So, for a 32 LED strip, there will be a total of 24*32 or 768 codes.

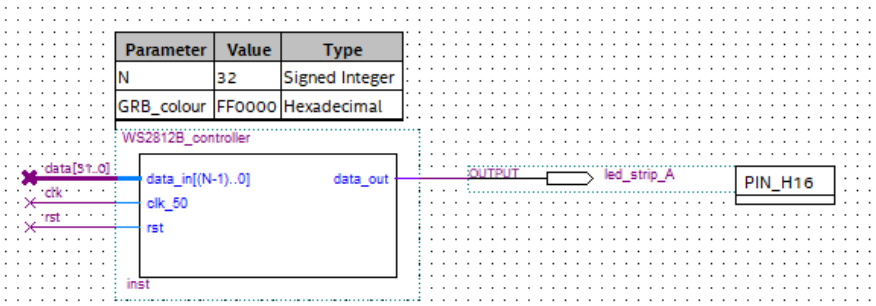


Figure 176 – a controller used for a 32 LED strip.

The following testbench simulates 3 LEDs being connected. Initially, the first two LEDs are to be lit up. Then, only the last LED is to be lit. The colour simulated is green (FF 00 00).

```
// testing
initial begin

    // rst
    rst = 1;
    data_in = 3'b110;
    #1us;

    rst = 0;

    #500us;       // account for possible reset code initially
    #86400ns;   //24*(400ns+800ns)*3LEDs

    #1000us;    // test eot condition

    // data change
    data_in = 3'b001;
    #500us;
    #86400ns;

    #1000us;    // test eot condition


    $stop;


end
```

Figure 177 – controller testbench code.

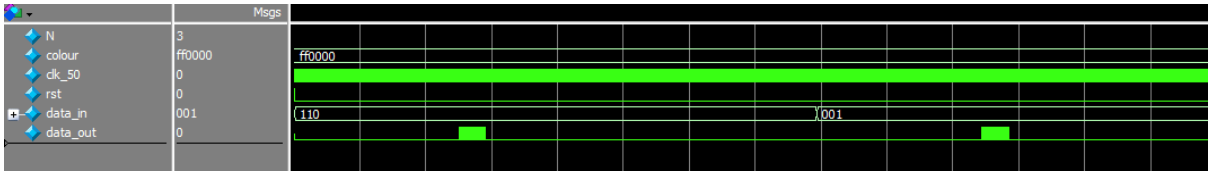


Figure 178 – testbench results. Note the two data frames being sent. One for **D1,D2,!D3** and one for **!D1,!D2,D3**.

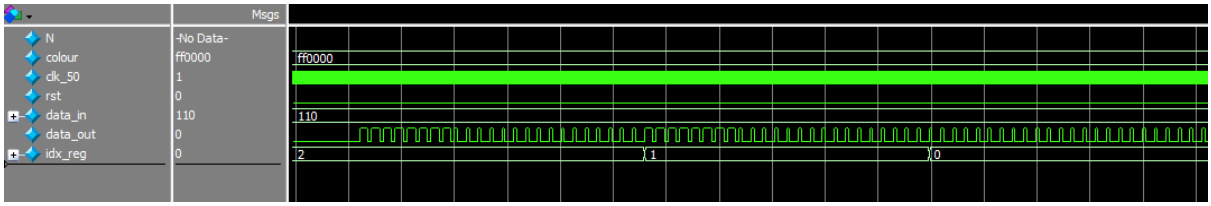


Figure 179 – zoomed in view on first data frame. idx_reg = 2, 1 and 0 represent LEDs D1, D2 and D3 respectively.

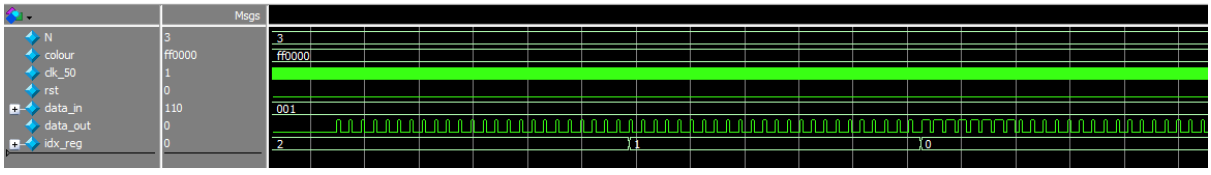


Figure 180 – zoomed in view on second data frame.

Important Note: *due to the reset frame being 500us, the fastest rate at which new data can be sent to the LEDs is approximately* $\frac{1}{500us} = 2kHz$*. This is way more than required.*

8.7.3 PCB Design

This PCB was to be a connector panel that interfaces the FPGA IO pins to the addressable LED strips. There were two iterations of this PCB.

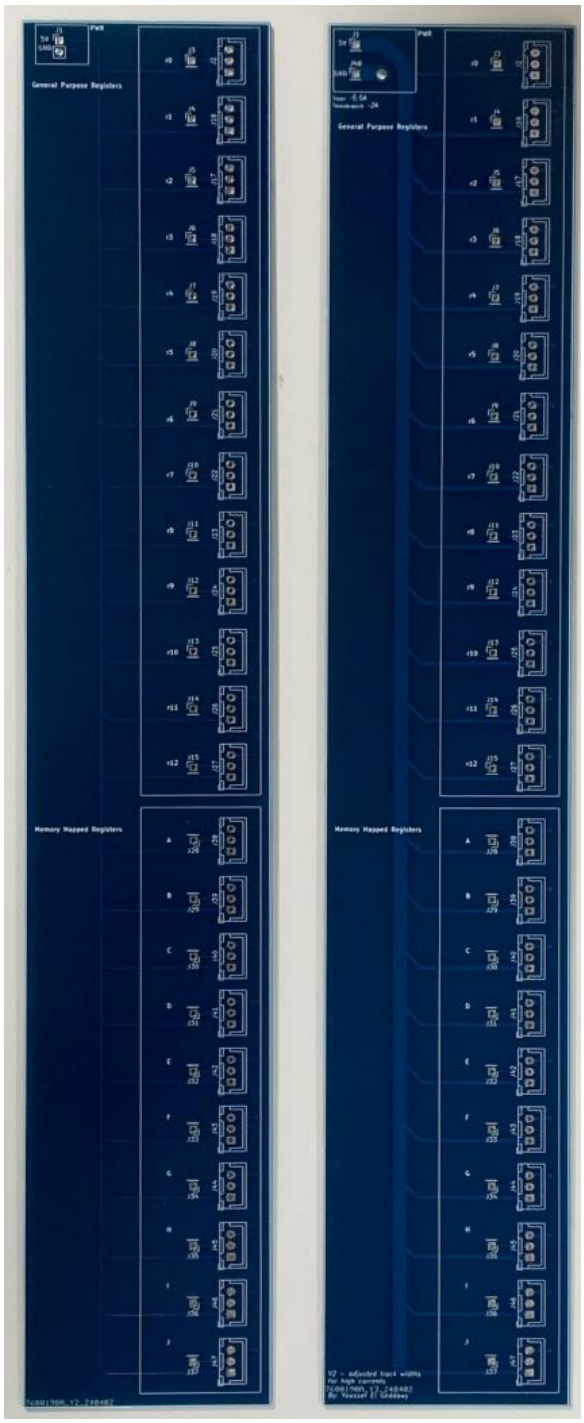


Figure 181 – PCB iteration A (left) and B (right). The first iteration used a default track width of 0.200 mm all round as well as tiny vias, whereas the second iteration uses 3.000 mm for the power rails; 0.760 mm for the individual LED strip power lines and 0.200 mm for the data lines in addition to much larger vias.

Both PCBs were two layers with the top layer being PWR and the bottom layer being GND.

Critical Problem – Track Widths, Vias and Maximum Current Flow

Due to the innate resistance of wires/PCB traces, there is an absolute maximum current rating that a PCB trace can carry given a certain width. If exceeded, the trace will become too hot and may start melting the solder mask or even the trace itself – a fire hazard! The table below gives a rough estimate of current capacity with varying trace widths.

IPC Recommended Track Width For 1 oz cooper PCB and 10 °C Temperature Rise

Current/A	Track Width(mil)	Track Width(mm)
1	10	0.25
2	30	0.76
3	50	1.27
4	80	2.03
5	110	2.79
6	150	3.81
7	180	4.57
8	220	5.59
9	260	6.60
10	300	7.62

Figure 182 – PCB track widths and current ratings [23].

Vias: are holes dug through from a top layer to a bottom layer to allow easy routing (and other reasons as well that are beyond the scope of this text and this project e.g. minimising loop areas; inductances and so on). But how big should they be? There are two dimensions of concern here the *via diameter* and the *via hole*. The general rule is:

The via diameter should be slightly greater than the trace width, and the via hole should be half the via diameter.

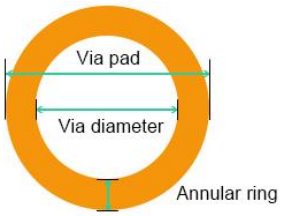


Figure 183 – Via dimensions labelled. *N.B. "Via pad" is also referred to as "via hole".*

All this is important as the addressable LEDs draw a lot more current than normal LEDs, especially at maximum brightness and more than one colour at a time (e.g. red and blue to make purple).

LED Characteristics

Parameter	Symbol	Color	Quiescent Current				Test Condition (Working current)
			Min.	Typ.	Max.	Unit	
Luminous intensity	IV	RED	300	--	500	mcd	16mA
		GREEN	800	--	1500		
		BLUE	200	--	300		
Wavelength	λd	RED	620	--	630	nm	16mA
		GREEN	515	--	525		
		BLUE	465	--	475		

Figure 184 – current draw is ~20mA per colour at maximum brightness [22].

The total number of LEDs required is:

$$13\ GPRs\ *\ 32\ LEDs\ =\ 416\ LEDs$$

$$10\ MMRs * 16\ LEDs = 160\ LEDs$$

$$Total\ LEDs = 416 + 160 = 576\ LEDs$$

The total current consumption can be calculated as follows (assuming all 576 LEDs are lit a single colour at half brightness):

$$\therefore \boldsymbol{I_{total} = 576 * \frac{20\ mA}{2} = 5.76A}$$

Recall that the original trace width for all traces (including the power rails) were 0.200 mm. Referring back to Figure 182, this means that the **first PCB iteration was rated for a maximum of less than 1A of current!**

Needless to say, this would be hazardous or at the very least, would not be sufficient to power all the LEDs at an acceptable brightness if a current limited PSU were to be used for safety. Hence a second iteration was designed and used, with appropriate trace widths, and testing proceeded.

To test the LED panel, mock register data was used via the “LPM_CONSTANT” component in Quartus.

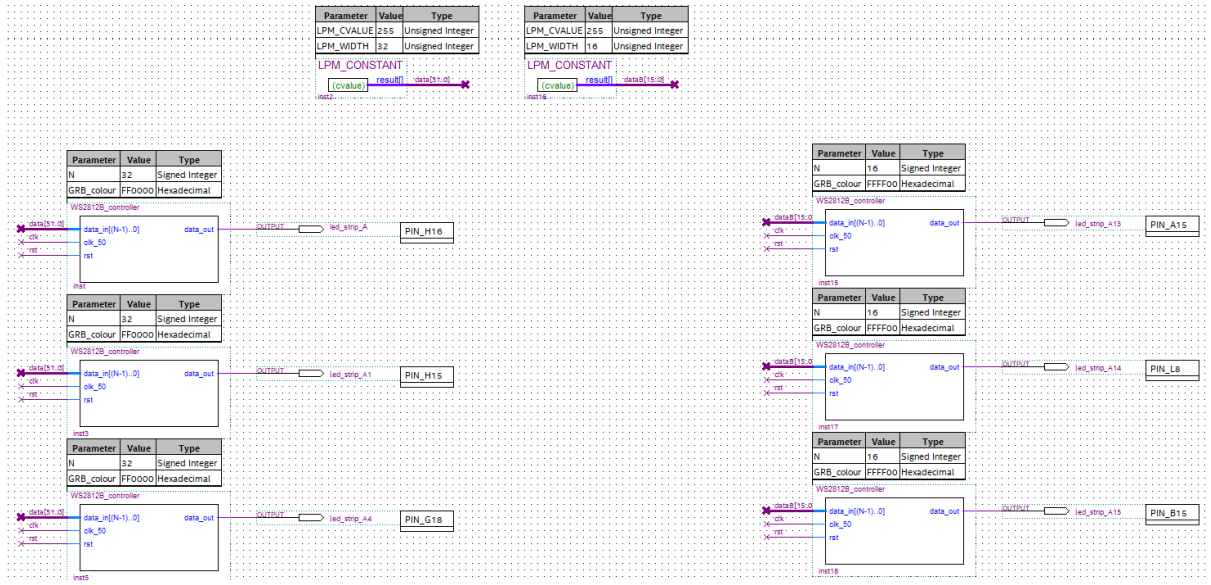


Figure 185 – connected the mock GPRs (green, left) and MMRs (yellow, right); all have values of 255. Only three of each are shown.

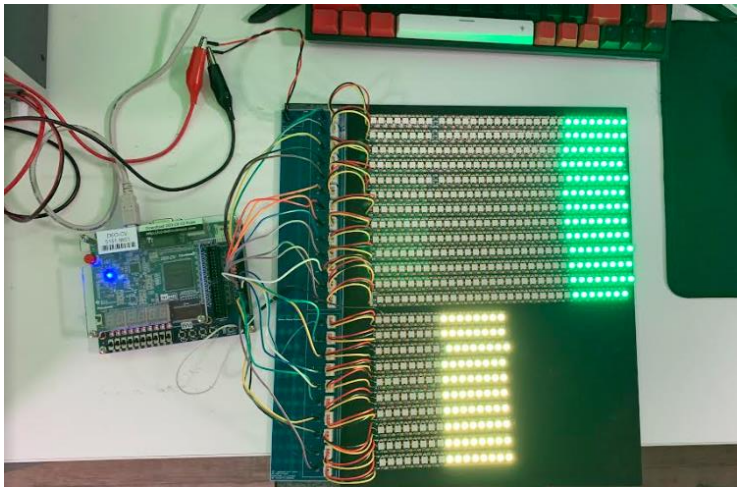


Figure 186 – **184 LEDs** power up test.

Figure 187 – current draw (10.55W of power).

9 Testing the Computer

Each notable component that makes up the computer was tested extensively. With so many modules working together eventually, the testbenches themselves must increase in size. With this, and before revealing the test results, there are a few notable things that were learned over time regarding testing:

- **Testbench accuracy:** it is important for the programmer to model the working behaviour of a circuit as accurately as possible to the real world. This may include things like simulating lines being idle before data being sent, or simulating clock frequencies accurately (not just choosing an arbitrary clock frequency), or even mocking multiple key presses in rapid succession. If testbenches are modelled accurately, there is a very high chance that if it works in simulation, it will work when synthesised on the FPGA; debugging in simulation is much easier than debugging hardware when synthesised.

- **Useful assertions:** with very large testbenches, the use of an error flag with assertions is very useful. It was found that, as the testbenches grew in complexity and size, printing out when tests are successful becomes a nuisance as this is not what the tester needs to know. Only printing out when assertions fail is more informative. The use of an error flag that is set whenever a test fails allows for a final conditional $display statement that prints to the terminal whether all tests have passed or not – useful information at a glance.

- **Automated testing:** naturally, testbenches will have to be repeated many times until all tests eventually pass. So, saving time and running all these tests automatically is invaluable. This is done using a file known as a *do file*. These files are scripts that, contain ModelSim commands that load all the desired settings automatically.

- **Type definitions (typedef) and testing internal signals:** it is possible to test all signals that are internal to the module itself. This is because Verilog syntax allows hierarchical naming systems e.g. dut.reg_internal. This can be taken advantage of by using typedefs; checking what state the component is in during a testbench written in plain English instead of numbers reduces cognitive load and is more understandable.

```
rst = 1;
data_in = 1;
clk = 0;
#10ns;

// idle for a bit
rst = 0;
data_in = 1;
#100ns;
```

Figure 188 – example of accurate modelling of line being idle before data transmission (UART).

```
// makes debugging easier when testbenching
// fetch1=6'd0 not fetch1=0, since that is implied as fetch1=32'd0!!
typedef enum logic [5:0] {

    fetch1=6'd0,fetch2,fetch3,nop1,mov1,ALTmov1,ALTmov2,ldr1,ldr2,ALTldr1,ALTldr2,ALTldr3,ALTldr4,str1,str2,str3,str4,
    ALTstr1,ALTstr2,ALTstr3,ALTstr4,cmp1,b1,bgt1,blt1,beq1,add1,add2,sub1,sub2,mul1,mul2,lsr1,lsr2,and1,and2,or1,or2,mvn1,mvn2

} CPUstate_t;
```

Figure 189 – example of using typedefs to make debugging easier.

```
// keep track of assertion fails
int unsigned err;


// testing
initial begin

    $display("\n\n");

    // rst
    rst = 1;
    #7ns;
    assert(dut.b2v_inst27.CPUstate == fetch1) else begin $error("failed rst"); err++; end
    #3ns;
```

Figure 190 – illustrating the use of error flags and testing internal signals (using typedefs).

```
# quits any running simulation
quit -sim

# (re)compiles verilog code
vlog CU_logic.sv
vlog CU_logic_tb.sv

# starts simulation with desired module testbench
vsim CU_logic_tb

# Set waveform configuration to show only leaf names
config wave -signalnamewidth 1

# adding an internal signal
add wave CU_logic_tb/dut/CPUstate

# Add input signals to waveform window
add wave -group "Inputs" N Z C V CPU_state

# Add output signals to waveform window
add wave -group "Outputs" RSELLOAD ROP1LOAD ROP2LOAD TRLOAD ARLOAD PCLOAD DRLOAD ACLOAD IRLOAD GPRLOAD MEMLOAD PCINC COUNTER_LD COUNTER_INC COUNTER_CLR
add wave -radix unsigned -group "Outputs" ALUSEL SYSTEMBUSSEL

# Run simulation
run -all

# Zoom to full waveform window
wave zoom full
```

Figure 191 – example of a do file (series of ModelSim commands – refer to the ModelSim command manual [24]).

With all that being said, the testbench results are shown starting on the next page.

The *flags_setter* module

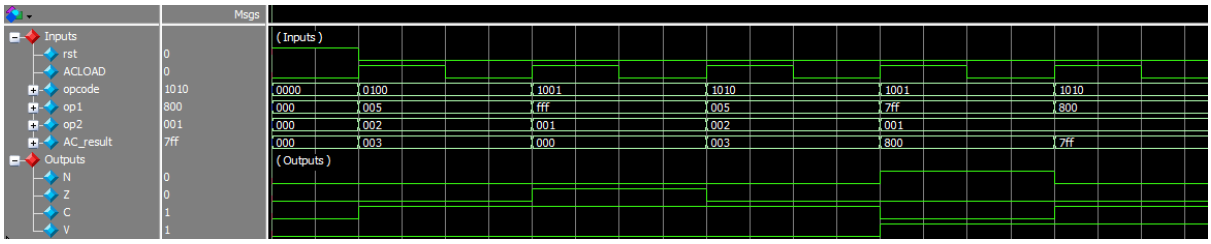


Figure 192 – testing the NZCV flags operation.

In order, the testbench includes:

1) **0100 (cmp):** $5 - 2 = 3$ **(C)**
2) **1001 (add):** $0xFFF + 0x001 = 0x000$ **(Z, C)**
3) **1010 (sub):** $5 - 2 = 3$ **(C)**
4) **1001 (add):** $0x7FF + 0x001 = 0x800$ **(N, V)**
5) **1010 (sub):** $0x800 - 0x001 = 0x7FF$ **(C, V)**

This was cross checked with an ARMv7 assembler [25]

```
1 .global _start
2 _start:
3
4     mov r0, #5
5     mov r1, #2
6     cmp r0, r1
```

```
cpsr | 20000ld3  NZCVI SVC
```

Figure 193 – test 1.

```
1 .global _start
2 _start:
3
4     mvn r0, #0
5     mov r1, #1
6     adds r2, r0,r1
```

```
cpsr | 60000ld3  NZCVI SVC
```

Figure 194 – test 2.

```
.global _start
_start:

    mov r0, #5
    mov r1, #2
    subs r2, r0,r1
```

```
cpsr | 200001d3  NZCVI SVC
```

Figure 195 – test 3.

```
.equ value, 0x7fffffff
.global _start
_start:

    ldr r0, =value
    mov r1, #1
    adds r2, r0,r1
```

```
cpsr | 900001d3  NZCVI SVC
```

Figure 196 – test 4.

```
.equ value, 0x80000000
.global _start
_start:

    ldr r0, =value
    mov r1, #1
    subs r2, r0,r1
```

```
cpsr | 300001d3  NZCVI SVC
```

Figure 197 – test 5.

(see next page)

The *GPR selector* module

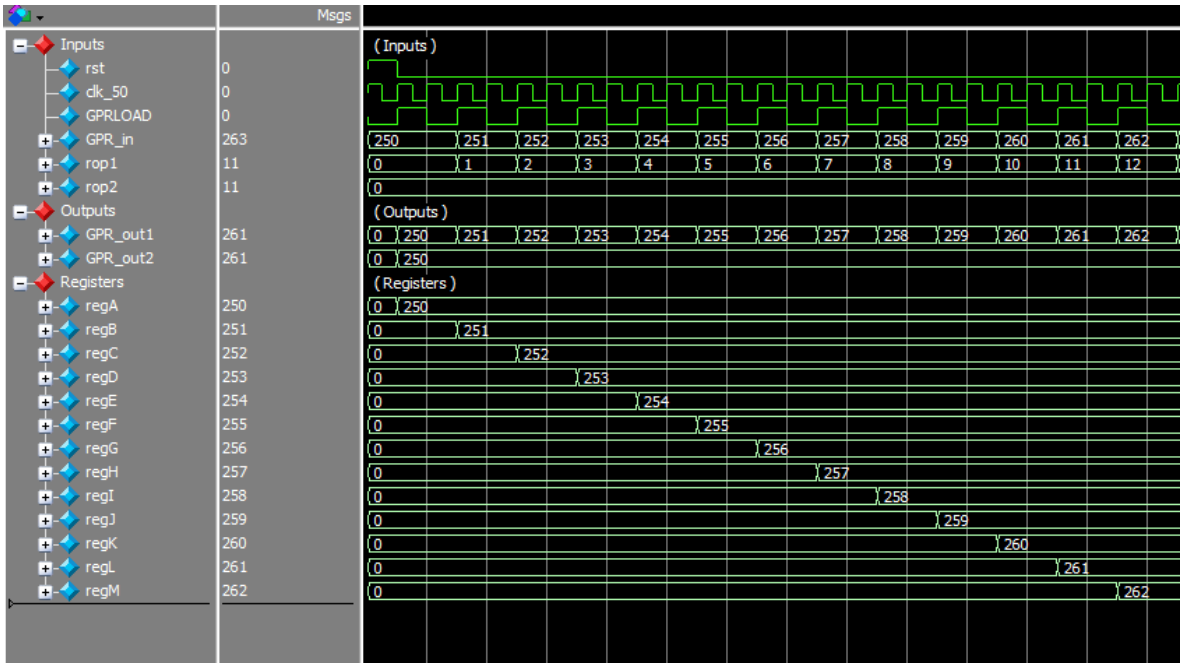


Figure 198 – loading data into the general-purpose registers, r0-r12.

The above test simulates loading values from 250 up to 262 in the GPRs r0-r12 respectively. The corresponding GPR is only loaded when the GPRLOAD signal is asserted. Note how the select line for the internal DEMUX-MUX pair comes from register rop1. Note how GPR_out1 is controlled by rop1 as well, whereas GPR_out2 is controlled by rop2 (not shown in the above waveform).

The ALU

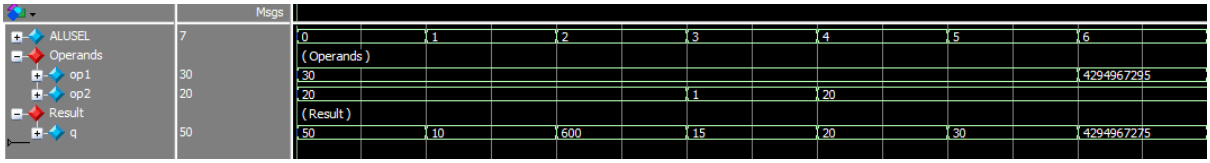


Figure 199 – ALU testbench. In order of: add, sub, mul, div, and, or, mvn.

The *operand_MUX* module that is connected to the ALU is shown as well.

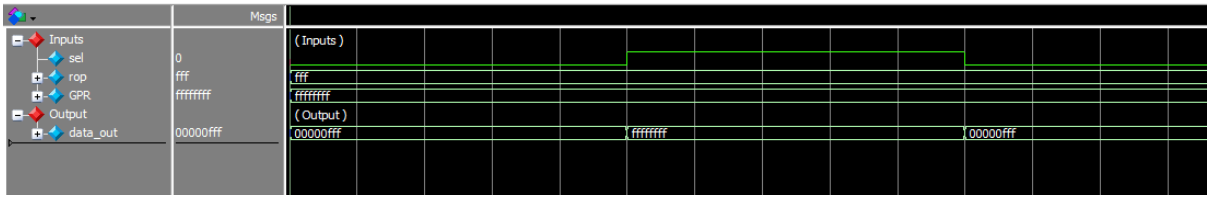


Figure 200 – *operand_MUX* testbench. If sel is low, rop is passed as the operand (literal), otherwise the contents of a register is passed as the operand.

The Control Unit

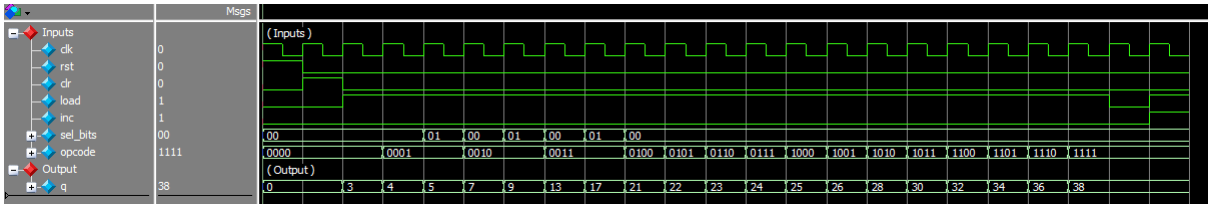


Figure 201 – the *CU_counter* testbench results.

Referring to Figure 59, the reader should observe the testbench results below.

```
# passed reset (FETCH1)
# passed clear (FETCH1)
# passed nopl
# passed movl
# passed movl
# passed ldrl
# passed ALTldrl
# passed strl
# passed ALTstrl
# passed cmpl
# passed bl
# passed bgtl
# passed bltl
# passed beql
# passed addl
# passed subl
# passed mull
# passed lsrl
# passed andl
# passed orl
# passed mvnl
# passed latching 1
# passed latching 2
```

Figure 202 – assertion passes for the state counter. Note the select bits and ALT execution routines.

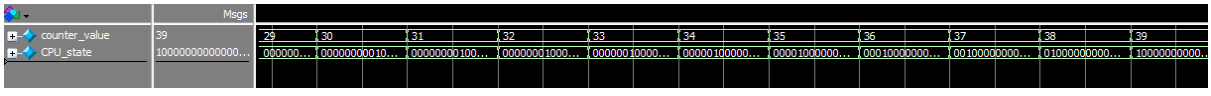


Figure 203 – last few states for the *CU_decoder* module.

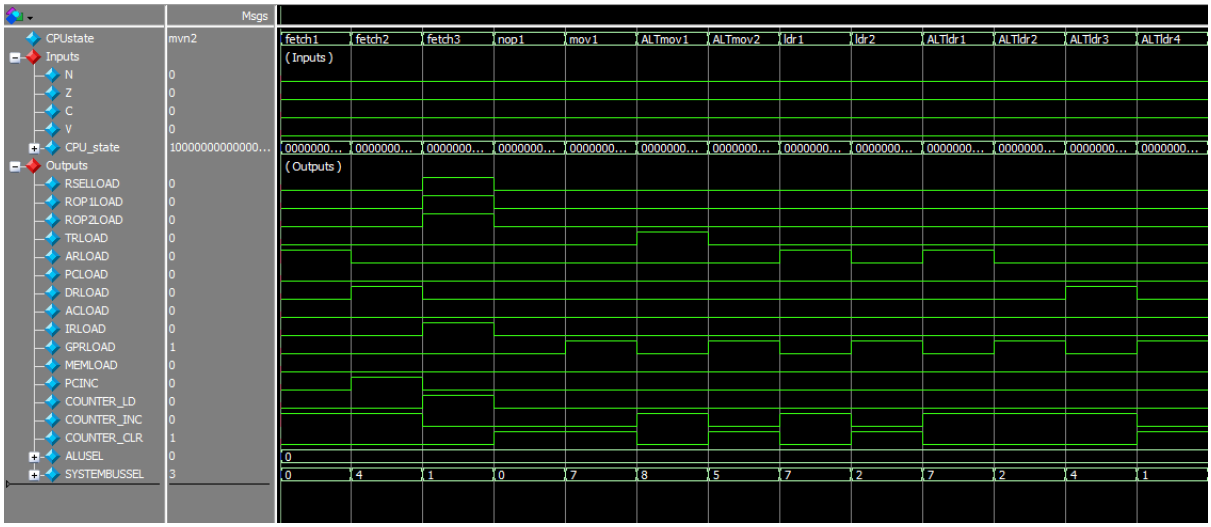


Figure 204 – snippet of the *CU_logic* testbench. *Refer to Figures 63 – 68.*

The Computer

With the CPU fully tested, a test instruction ROM was connected as its instruction memory. To test the computer fully, this test ROM contained every single instruction combination possible, including all the ALT instructions, and working with all the GPRs and MMRs. *In total, the testbench file came in at about 1900 lines of code, taking about 2 days just to complete the testbench. Countless bugs were discovered and squashed during this period.*

```
// Generally, the word structure is: OPCODE s[1..0] a[15..0] b[15..0]
// "bit" datatype to avoid z and x when address is out of bounds
bit [DATA_WIDTH-1:0] rom [0:number_of_test_instructions-1] = '{

    /////// nop ///////
    {4'b0000,2'b00,16'h0000,16'h0000},      // 0 : nop

    /////// mov ///////
    {4'b0001,2'b00,16'h0000,16'h0001},      // 1 : mov r0,   #1
    {4'b0001,2'b00,16'h1000,16'h0002},      // 2 : mov r1,   #2
    {4'b0001,2'b00,16'h2000,16'h0003},      // 3 : mov r2,   #3
    {4'b0001,2'b00,16'h3000,16'h0004},      // 4 : mov r3,   #4
    {4'b0001,2'b00,16'h4000,16'h0005},      // 5 : mov r4,   #5
    {4'b0001,2'b00,16'h5000,16'h0006},      // 6 : mov r5,   #6
    {4'b0001,2'b00,16'h6000,16'h0007},      // 7 : mov r6,   #7
    {4'b0001,2'b00,16'h7000,16'h0008},      // 8 : mov r7,   #8
    {4'b0001,2'b00,16'h8000,16'h0009},      // 9 : mov r8,   #9
    {4'b0001,2'b00,16'h9000,16'h000A},      // 10 : mov r9,  #10
    {4'b0001,2'b00,16'hA000,16'h000B},      // 11 : mov r10, #11
    {4'b0001,2'b00,16'hB000,16'h000C},      // 12 : mov r11, #12
    {4'b0001,2'b00,16'hC000,16'h000D},      // 13 : mov r12, #13

    {4'b0001,2'b01,16'h0000,16'h1000},      // 14 : mov r0,  r1
    {4'b0001,2'b01,16'h1000,16'h2000},      // 15 : mov r1,  r2
    {4'b0001,2'b01,16'h2000,16'h3000},      // 16 : mov r2,  r3
    {4'b0001,2'b01,16'h3000,16'h4000},      // 17 : mov r3,  r4
    {4'b0001,2'b01,16'h4000,16'h5000},      // 18 : mov r4,  r5
    {4'b0001,2'b01,16'h5000,16'h6000},      // 19 : mov r5,  r6
    {4'b0001,2'b01,16'h6000,16'h7000},      // 20 : mov r6,  r7
    {4'b0001,2'b01,16'h7000,16'h8000},      // 21 : mov r7,  r8
    {4'b0001,2'b01,16'h8000,16'h9000},      // 22 : mov r8,  r9
    {4'b0001,2'b01,16'h9000,16'hA000},      // 23 : mov r9,  r10
    {4'b0001,2'b01,16'hA000,16'hB000},      // 24 : mov r10, r11
    {4'b0001,2'b01,16'hB000,16'hC000},      // 25 : mov r11, r12
    {4'b0001,2'b01,16'hC000,16'h0000},      // 26 : mov r12, r0
```

Figure 205 – the first 27 test instructions in the test ROM.

```
/////// add ///////
{4'b0001,2'b00,16'h0000,16'h000B},      // 144 : mov r0, #11
{4'b0001,2'b00,16'h1000,16'h000A},      // 145 : mov r1, #10

{4'b1001,2'b00,16'h200B,16'h000A},      // 146 : add r2, #11,#10
{4'b1001,2'b01,16'h200B,16'h0000},      // 147 : add r2, #11,r0
{4'b1001,2'b10,16'h2100,16'h000A},      // 148 : add r2, r1,#10
{4'b1001,2'b11,16'h2000,16'h0100},      // 149 : add r2, r0,r1

/////// sub ///////
{4'b1010,2'b00,16'h200B,16'h000A},      // 150 : sub r2, #11,#10
{4'b1010,2'b01,16'h200B,16'h0000},      // 151 : sub r2, #11,r0
{4'b1010,2'b10,16'h2100,16'h000A},      // 152 : sub r2, r1,#10
{4'b1010,2'b11,16'h2000,16'h0100},      // 153 : sub r2, r0,r1

/////// mul ///////
{4'b1011,2'b00,16'h200B,16'h000A},      // 153 : mul r2, #11,#10
{4'b1011,2'b01,16'h200B,16'h0000},      // 155 : mul r2, #11,r0
{4'b1011,2'b10,16'h2100,16'h000A},      // 156 : mul r2, r1,#10
{4'b1011,2'b11,16'h2000,16'h0100},      // 157 : mul r2, r0,r1

/////// lsr ///////
{4'b1100,2'b10,16'h2100,16'h0001},      // 158 : lsr r2, r1,#1

/////// and ///////
{4'b1101,2'b00,16'h200B,16'h000A},      // 159 : and r2, #11,#10
{4'b1101,2'b01,16'h200B,16'h0000},      // 160 : and r2, #11,r0
{4'b1101,2'b10,16'h2100,16'h000A},      // 161 : and r2, r1,#10
{4'b1101,2'b11,16'h2000,16'h0100},      // 162 : and r2, r0,r1

/////// or ///////
{4'b1110,2'b00,16'h200B,16'h000A},      // 163 : or r2, #11,#10
{4'b1110,2'b01,16'h200B,16'h0000},      // 164 : or r2, #11,r0
{4'b1110,2'b10,16'h2100,16'h000A},      // 165 : or r2, r1,#10
{4'b1110,2'b11,16'h2000,16'h0100},      // 166 : or r2, r0,r1

/////// mvn ///////
{4'b1111,2'b00,16'h2000,16'h0001},      // 167 : mvn r2, #1
{4'b1111,2'b01,16'h2000,16'h1000}       // 168 : mvn r2, r1
```

Figure 206 – testing the AL instructions via the test ROM.

Problems*:

- During testing, it was found that the output of the three registers rsel, rop1 and rop2 lagged a clock cycle behind. This caused synchronisation problems. There were two simple fixes to this.
- The first potential fix was to use an asynchronous load line. The problem with this (ignoring metastability) is that, from experience, it was discovered that Quartus fails to synthesise asynchronous logic sometimes (characterised by weird floating outputs that can be seen using some kind of LED display). Another approach was required.
- The other solution was to change them from flip-flops to latches instead (using the always_latch construct rather than always_ff). This solution was simple and avoided the use of asynchronous logic. This fixed the timing issues and later synthesised successfully!

** An example of one of the bugs encountered. There were many more…*

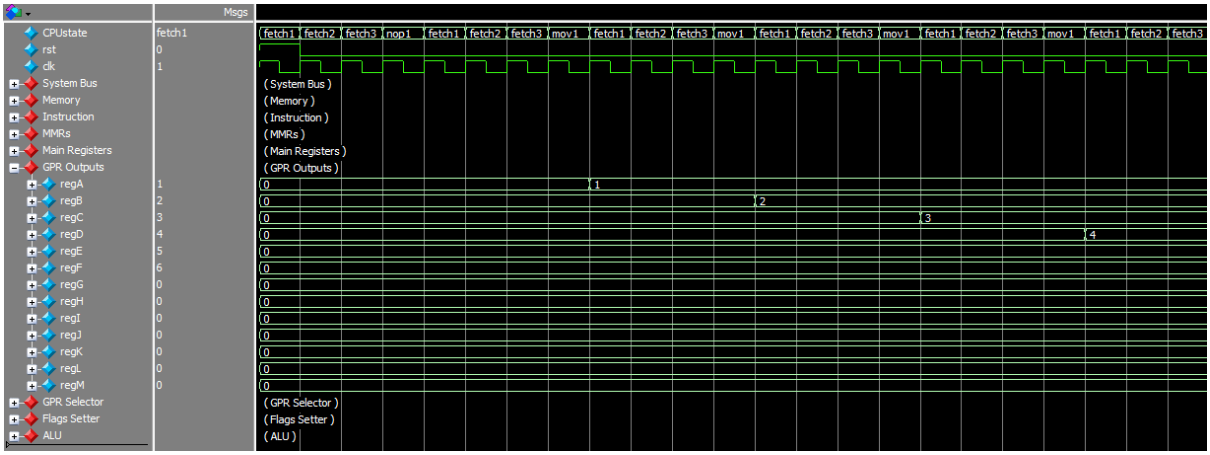


Figure 207 – snippet of the computer running. Note the 3-clock cycle FETCH in between each instruction execution.

The instructions that the CPU fetched in the above waveform are (refer to Figure 205):

1) nop
2) mov r0, #1
3) mov r1, #2
4) mov r2, #3
5) mov r3, #4

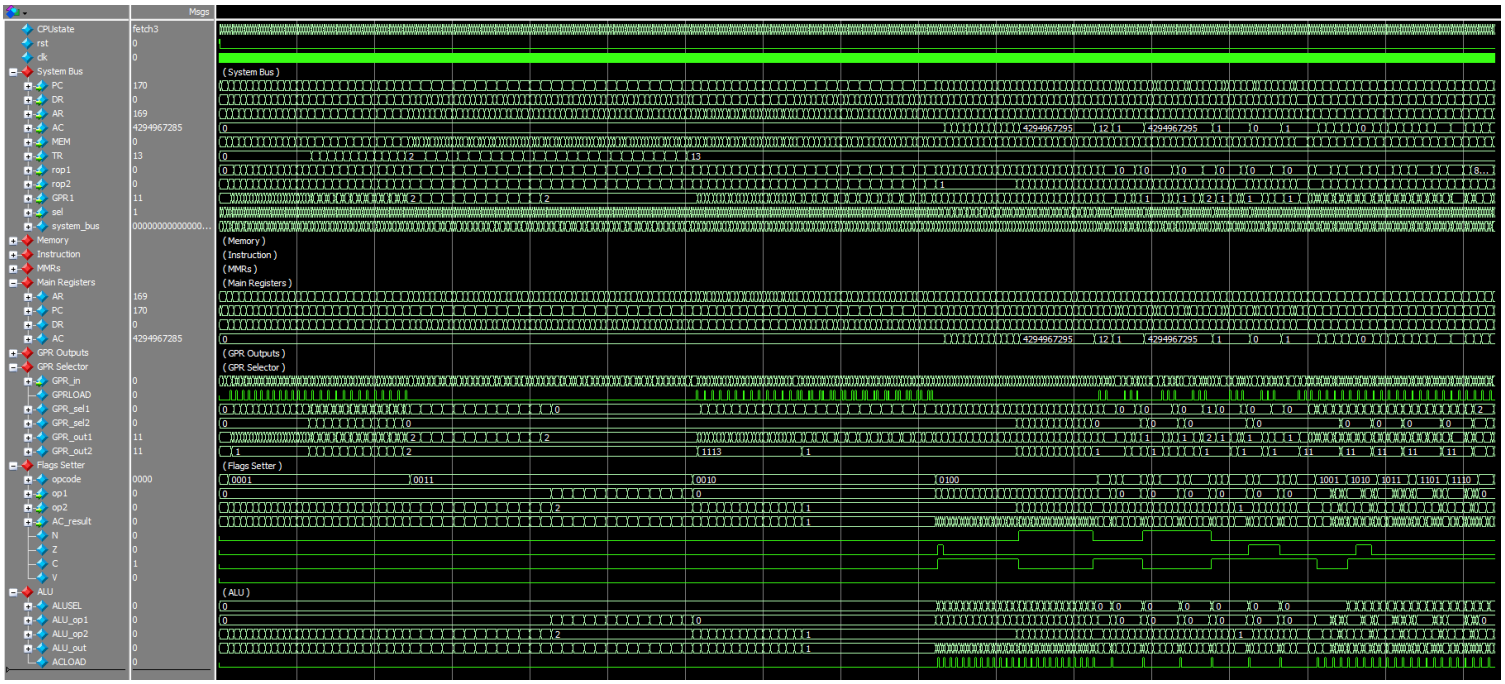


Figure 208 – The computer running all 169 instructions in the test ROM.

The above Figure cannot be understood by the reader, that is not its purpose. Rather, the author's intent was to show the beautiful complexity of this intricate machine. Everything running in a synchronised and harmonious manner – nothing a clock cycle too late; nothing a clock cycle too early.

10 The Assembler

This was the simplest and most straightforward part of the project. The idea was to use a MCU's UART to send the instruction bits and program the CPU. This was done using C++ functions. C++ was chosen over C for one essential reason: **function overloading.** It is a language feature in C++ that allows the redefinition of the same function so long as they differ in arguments. This was crucial to allow implementation of multiple forms of the same assembly instruction. An example is shown below:

```
// example: mov(r0, 5)
void mov(gpr_t gpr, uint16_t literal) {

    opcode = 0x01;
    sel = 0x00;
    op1 = gpr;
    op2 = literal;

    transmit_instruction();

}
// example: mov(r0, r1)
void mov(gpr_t gpr1, gpr_t gpr2) {

    opcode = 0x01;
    sel = 0x01;
    op1 = gpr1;
    op2 = gpr2;

    transmit_instruction();

}
```

Figure 209 – example of how function overloading can be used with the mov instruction.

The assembler has one very important language feature: **variable support.** First, an assembly function called word was implemented. This was inspired by ARM's .word directive, which stores a 32-bit variable on the stack.

```
// variables section
.data
num1:
    .word 12

num2:
    .word 15

result:
    .word 0

test:
    .word 1
```

Figure 210 – variable support in ARM assembly.

```
// example: word("myvar", 25)
void word(string var_name, uint16_t var_value) {

    var_names[str_idx] = var_name;
    var_values[str_idx] = var_value;
    stack_addr[str_idx] = 0x44C + str_idx; // boundary address for stack mem is 0x44C

    // implicitly calls str
    str(var_value,stack_addr[str_idx]);

    str_idx++;



}
```

Figure 211 – implementation of variable support in the assembler. Note the implicit call to the str instruction.

Next, the assembler needed to know information about the stack memory. Free addresses; variables stored in addresses along with their values and names.

```
// stack releated section
string var_names[2996];
uint16_t var_values[2996];
uint16_t stack_addr[2996]; // stack address counter; starts at boundary address of stack memory
uint16_t str_idx = 0;
uint16_t ldr_idx = 0;
```

Figure 212 – using arrays to store variable names, addresses and values. Refer to Figure 73 (The Memory Map).

Each time a variable is stored on the stack using the word function, the str function is not only called but an index counter called str_idx is incremented as well. This index counter tells the assembler where next to store a variable on the stack. What about loading said variables?

```
// example: ldr(r0, num1)
void ldr(gpr_t gpr, string var_name) {

    for(int i=0; i<2996; i++) {

        if(var_names[i] == var_name) {

            ldr_idx = i;
            break;
        }
    }

    // implicity calls ldr
    ldr(gpr,"[]",stack_addr[ldr_idx]);


}
```

Figure 213 – loading the value of a variable into a GPR. Note the implicit call to the ldr instruction.

The entire assembler implementation, as well as example programs including the use of memory mapped peripherals can be found in the Appendix of this text.

Here is an example program demonstrating stack usage.

```
1   word("num1", 1);    // variables stored on stack
2   word("num2", 2);    //
3   ldr(r0, "num1");
4   ldr(r1, "num2");
5   add(r2, r0,r1);
```

Figure 214 – storing two variables num1 and num2 on the stack, loading them and storing their sum into r2.

```
--------- PROGRAMMING CPU... ---------
1: 0003, 01, 0001, 044c

2: 0003, 01, 0002, 044d

3: 0002, 01, 0000, 044c

4: 0002, 01, 1000, 044d

5: 0009, 03, 2000, 0100

////////////////////////////////////////////////////////

The variables stored on the stack are:
num1 num2

Their values:
1 2

Their addresses:
44c 44d

////////////////////////////////////////////////////////

--------- PROGRAMMING COMPLETE! ---------
```

Figure 215 – programming process shown on terminal. Note the stack usage report at the end.

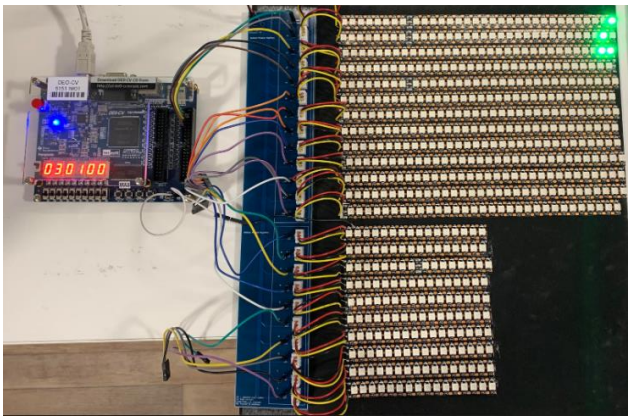


Figure 216 – register contents correct!

When coding example programs in assembly, the need for delays was very apparent. So, **hardware timers** were implemented. These timers would be memory mapped to allow control of them using assembly instructions. *See the Appendix for more details.*

```
module CPUTimer (

    output logic rdy,

    input logic [12:0] Tms,
    input logic clk, rst
);
```

Figure 217 – *CPUTimer* module port definitions. The CPU asserts a 12-bit time (in ms) on the Tms bus.

The memory mapped connections are shown below.

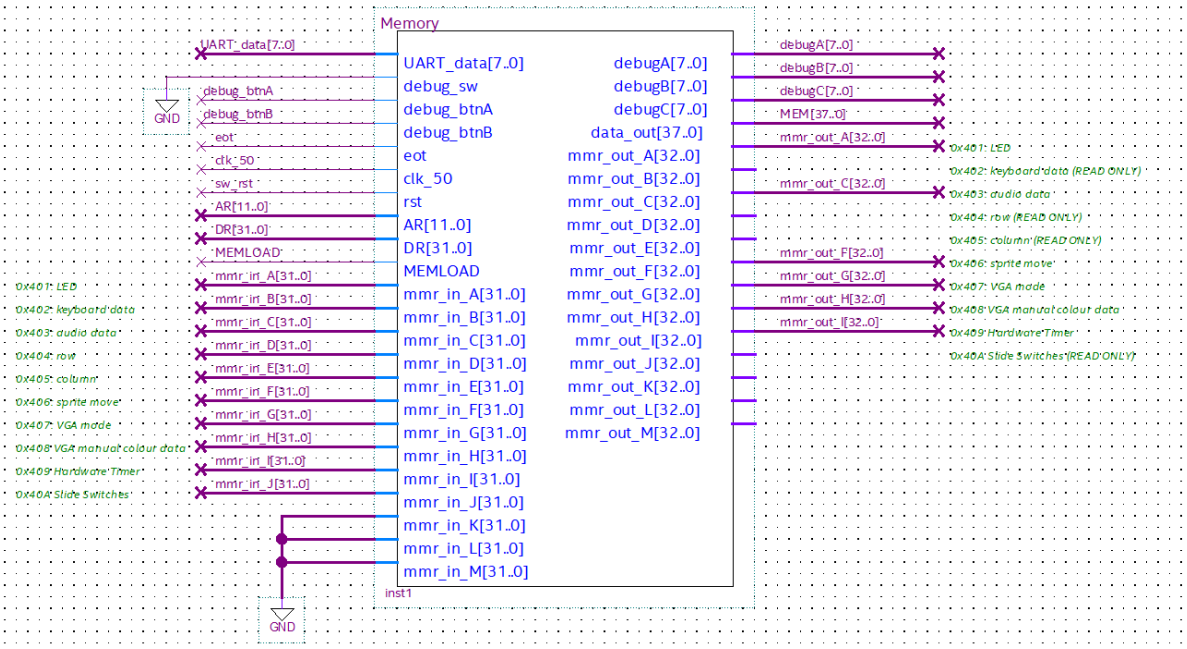


Figure 218 – Memory mapped peripheral registers connected to the CPU's memory.

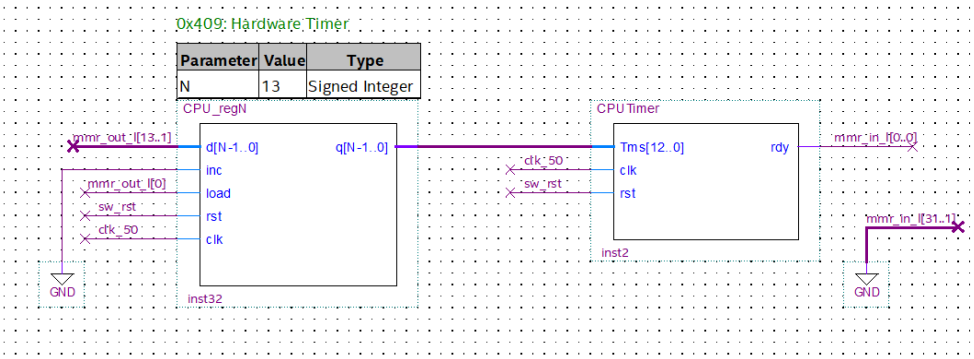


Figure 219 – CPU hardware timer connected to an MMR.

N.B. to fit the design using the FPGA's limited resources, the following adjustments had to be made:

- ***Instruction RAM:*** *limited to 100 instructions*
- ***Stack memory:*** *limited to 20 memory locations (variables)*

This was done as BRAM inference optimisations were not made in this project.

11 Future Development and Improvements

A list of improvements that may be made in the future to make this project more user friendly and optimised is listed in this section.

- **Logic utilisation optimisation:** reducing the number of FPGA resources used by reducing unnecessarily large register widths and using memory IP cores to infer BRAM. This will allow the project to fit in smaller FPGAs and lower compilation times.

- **Assembler label support:** the latest version of the assembler does not support labels. This makes using branch statements frustrating as the instruction number needs to be known. Currently, an assembly program is written in an empty text file with line numbering enabled as a temporary workaround.

- **Extended assembler variable support:** whilst the current assembler supports storing variables in memory, these variables act more like constants and cannot be updated. Furthermore, initialisation of variables with register contents may be implemented.

- **PCB adjustments (v3):** the use of a single 23-pin header rather than 23 separated jumper cables would improve the cable management and make it more pleasing to the eye. In addition, the silkscreen text could be made larger to improve legibility.

- **Graphics rendering capabilities:** the CPU's control over the graphics controller is very simple and is only a proof of concept. The hardware and assembler may be adjusted to accommodate for more intricate control (e.g. specialised instructions) that allow rendering of geometry on the screen.

12 Conclusion

As a concluding statement regarding this project, the author would like to share personal insights.

This project proved to be difficult yet greatly improved the skills of the author, particularly in RTL design. One notable thing to mention is that development in general is very time consuming, but hardware design is even more time consuming. Very little time is spent coding, and, unlike software development, a single code file written in HDL is typically small (usually around 50-300 lines of code). This is because hardware design requires many small parts to work in tandem with each other. In fact, most of the time is spent planning the design, then writing testbenches, simulating and debugging the hardware: an iterative process until ultimately, all the tests pass.

When compared to software programming languages (e.g. C or Python), the concepts that need to be learned in an HDL are very limited – there aren't many concepts that need to be known to master the language. But the concepts that do exist need to be known very well to ensure the hardware behaves as intended.

A CPU is a complicated structure with many working components. Testing large circuitry like this requires a smart approach; an incremental one, where tests are done on small modules first before connecting them and increasing the test size. Shortcuts should not be taken when writing testbenches! A well written testbench is usually time consuming and requires a bit of effort. This pays off however, allowing for bugs and design flaws to be caught relatively easily. A poorly written testbench will not reveal all the design errors, leaving the user scratching their head wondering why the circuitry does not work as intended.

Despite the fear that is project may be overly ambitious to have completed within 3 months, it was eventually completed to a very high standard and has all of the features that were intended to be present in the final product.

13 References

[1] https://vanhunteradams.com/DE1/VGA_Driver/Driver.html

[2] https://martin.hinner.info/vga/timing.html

[3] https://www.edaboard.com/threads/what-are-hsync-and-vsync.192041/

[4] https://talk.macpowerusers.com/t/studio-monitor-backlight-bleed/28483

[5] https://www.pinterest.com/pin/763289836825368814/

[6] https://en.wikipedia.org/wiki/Video_Graphics_Array

[7] https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=163&No=921

[8] https://www2.pcs.usp.br/~labdig/material/DE0_CV_User_Manual.pdf

[9] https://web.mit.edu/6.111/www/f2017/serial/lab_serial.html

[10] https://www.st.com/en/evaluation-tools/nucleo-f429zi.html

[11] https://karooza.net/how-to-interface-a-ps2-keyboard

[12] https://mixbutton.com/mixing-articles/music-note-to-frequency-chart/

[13] https://www.analog.com/en/resources/technical-articles/selecting-and-using-rs232-interface-parts-for-your-power-supply-voltages.html

[14] https://www.intel.com/content/www/us/en/docs/programmable/683801/current/cyclone-v-device-datasheet.html

[15] https://www.quora.com/Why-havent-they-reduced-logic-values-in-semiconductors-to-1V-and-0V-instead-of-5V-It-would-save-power-reduce-heating-and-allow-for-even-greater-compactness

[16] https://developer.arm.com/documentation/ddi0439/b/CHDDIGAC

[17] https://developer.arm.com/documentation/den0042/a/Unified-Assembly-Language-Instructions/Instruction-set-basics/Conditional-execution

[18] https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/hdl/vlog/vlog_file_dir_ram.htm

[19] https://www.intel.com/content/www/us/en/docs/programmable/683694/current/maximum-resources-04257.html

[20] https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/glosslist.htm

[21] https://cdn-shop.adafruit.com/datasheets/WS2812B.pdf

[22] https://www.mouser.com/pdfDocs/WS2812B-2020_V10_EN_181106150240761.pdf

[23] https://www.mclpcb.com/blog/pcb-trace-width-vs-current-table/

[24] https://www.microsemi.com/document-portal/doc_view/136364-modelsim-me-10-4c-command-reference-manual-for-libero-soc-v11-7

[25] https://cpulator.01xz.net/

14 Appendix

A1 Assembler Functions Definitions

```
1. // functions file
2. #include "functions.hpp"
3. 
4. // USART2
5. UnbufferedSerial uart2(PD_5, PD_6);
6. 
7. // instruction counter
8. uint16_t instruction_cnt = 1;
9. 
10. // the 6 bytes to be transmitted
11. uint8_t opcode;     // uint8_t = unsigned char
12. uint8_t sel;
13. uint8_t op1h;
14. uint8_t op1l;
15. uint8_t op2h;
16. uint8_t op2l;
17. 
18. // the two "raw" operands
19. uint16_t op1;
20. uint16_t op2;
21. 
22. // stack releated section
23. string var_names[2996];
24. uint16_t var_values[2996];
25. uint16_t stack_addr[2996]; // stack address counter; starts at boundary address of stack memory
26. uint16_t str_idx = 0;
27. uint16_t ldr_idx = 0;
28. 
29. void programming_start() {
30.     printf("\033[38;5;229m"); // pale yellow
31.     printf("\n--------- PROGRAMMING CPU... ---------\n");
32. }
33. 
34. void programming_end() {
35.     printf("\n--------- PROGRAMMING COMPLETE! ---------\n");
36. }
37. 
38. void clear_terminal() {
39.     // clear terminal and move cursor to (0,0)
40.     printf("\033[2J"); printf("\033[H");
41. }
42. 
43. void stack_usage() {
44. 
45.     cout << "\n////////////////////////////////////////////////////////" << endl << endl;
46.     cout << "\nThe variables stored on the stack are:" << endl;
47. 
48.     for(int i=0;i<str_idx;i++) {
49. 
50.         cout << var_names[i] << " ";
51. 
52.     }
53. 
54.     cout << "\n\nTheir values: " << endl;
55. 
56.     for(int i=0;i<str_idx;i++) {
57. 
58.         cout << var_values[i] << " ";
59. 
60.     }
61. 
```

```
62.     cout << "\n\nTheir addresses: " << endl;
63.
64.     for(int i=0;i<str_idx;i++) {
65.
66.         printf("%03x ",stack_addr[i]);
67.
68.     }
69.
70.     cout << "\n\n////////////////////////////////////////////////////" << endl << endl;
71.
72.
73. }
74. void uart2_setup() {
75.
76.     uart2.baud(9600);    // WARNING: does not work properly below 9600 baud (e.g. 300 baud)
77.                          // tested using scope
78.     uart2.format(
79.         /* bits */ 8,
80.         /* parity */ SerialBase::None,
81.         /* stop bit */ 1
82.     );
83.
84. }
85.
86. void transmit(char ch) {
87.
88.     // wait until TXE bit is high (transferred data to shift register)
89.     // i.e. wait until Tx buffer is empty
90.     while(!(USART2->SR & 0x80)) {}
91.
92.     // places byte in DR
93.     // clears the TXE bit automatically
94.     USART2->DR = ch;
95.
96.     wait_us(10000);   // 10ms delay
97.
98. }
99.
100. void transmit_instruction() {
101.
102.     op1h = op1 >> 8;
103.     op1l = op1 & 0x00FF;
104.     op2h = op2 >> 8;
105.     op2l = op2 & 0x00FF;
106.
107.     transmit(opcode);
108.     transmit(sel);
109.     transmit(op1h);
110.     transmit(op1l);
111.     transmit(op2h);
112.     transmit(op2l);
113.
114.     print_instruction();
115. }
116.
117. void transmit_ALinstruction() {
118.
119.     transmit(opcode);
120.     transmit(sel);
121.     transmit(op1h);
122.     transmit(op1l);
123.     transmit(op2h);
124.     transmit(op2l);
125.
126.     print_instruction();
127. }
128.
129. void test_instruction(uint8_t opcd, uint8_t select, uint16_t oprnd1, uint16_t oprnd2) {
130.
131.     op1 = oprnd1;
132.     op2 = oprnd2;
```

```
133.
134.    opcode = opcd;
135.    sel = select;
136.    op1h = op1 >> 8;
137.    op1l = op1 & 0x00FF;
138.    op2h = op2 >> 8;
139.    op2l = op2 & 0x00FF;
140.
141.    print_instruction();
142.
143. }
144.
145. void print_instruction() {
146.
147.    printf("\n%d: ", instruction_cnt);
148.    printf("%04x, ",opcode);          // print 4 digits, with trailing zeros if needed
149.    printf("%02x, ",sel);
150.    printf("%02x",op1h);
151.    printf("%02x, ",op1l);
152.    printf("%02x",op2h);
153.    printf("%02x\n\n",op2l);
154.
155.    instruction_cnt++;
156. }
157.
158. // instructions
159. // will make heavy use of function overloading - redefinition of function
160. // with differing arguments
161.
162. // example: nop()
163. void nop() {
164.
165.    opcode = 0x00;
166.    sel = 0x00;
167.    op1 = 0x0000;
168.    op2 = 0x0000;
169.
170.    transmit_instruction();
171.
172. }
173.
174. // example: mov(r0, 5)
175. void mov(gpr_t gpr, uint16_t literal) {
176.
177.    opcode = 0x01;
178.    sel = 0x00;
179.    op1 = gpr;
180.    op2 = literal;
181.
182.    transmit_instruction();
183.
184. }
185. // example: mov(r0, r1)
186. void mov(gpr_t gpr1, gpr_t gpr2) {
187.
188.    opcode = 0x01;
189.    sel = 0x01;
190.    op1 = gpr1;
191.    op2 = gpr2;
192.
193.    transmit_instruction();
194.
195. }
196.
197. // example: str(r0, 0x44C)
198. void str(gpr_t gpr, uint16_t addr) {
199.
200.    opcode = 0x03;
201.    sel = 0x00;
202.    op1 = gpr;
203.    op2 = addr;
```

```
204.
205.     transmit_instruction();
206.
207. }
208. // example: str(2, 0x44C)
209. void str(uint16_t literal, uint16_t addr) {
210.
211.     opcode = 0x03;
212.     sel = 0x01;
213.     op1 = literal;
214.     op2 = addr;
215.
216.     transmit_instruction();
217.
218. }
219. // example: str(r0, vga_colour)
220. void str(gpr_t gpr, mmr_t mmr) {
221.
222.     if(mmr == keyboard_input || mmr == vga_row || mmr == vga_column) {
223.
224.         printf("\nERROR: ATTEMPTING TO WRITE TO READ ONLY REGISTER\n");
225.         sleep();
226.
227.     }
228.
229.     opcode = 0x03;
230.     sel = 0x00;
231.     op1 = gpr;
232.     op2 = mmr;
233.
234.     transmit_instruction();
235.
236. }
237. // example: str(2, vga_colour)
238. void str(uint16_t literal, mmr_t mmr) {
239.
240.     if(mmr == keyboard_input || mmr == vga_row || mmr == vga_column) {
241.
242.         printf("\nERROR: ATTEMPTING TO WRITE TO READ ONLY REGISTER\n");
243.         sleep();
244.
245.     }
246.
247.     opcode = 0x03;
248.     sel = 0x01;
249.     op1 = literal;
250.     op2 = mmr;
251.
252.     transmit_instruction();
253.
254. }
255. // example: word("myvar", 25)
256. void word(string var_name, uint16_t var_value) {
257.
258.     var_names[str_idx] = var_name;
259.     var_values[str_idx] = var_value;
260.     stack_addr[str_idx] = 0x44C + str_idx; // boundary address for stack mem is 0x44C
261.
262.     // implicitly calls str
263.     str(var_value,stack_addr[str_idx]);
264.
265.     str_idx++;
266.
267.
268.
269. }
270.
271. // example: ldr(r0, 0x44C)
272. void ldr(gpr_t gpr, uint16_t addr) {
273.
274.     opcode = 0x02;
```

```
    sel = 0x00;
    op1 = gpr;
    op2 = addr;

    transmit_instruction();

}
// example: ldr(r0, "[]", 0x44C)
void ldr(gpr_t gpr, string a, uint16_t deref_addr) {

    opcode = 0x02;
    sel = 0x01;
    op1 = gpr;
    op2 = deref_addr;

    transmit_instruction();

}
// example: ldr(r0, vga_colour)
void ldr(gpr_t gpr, mmr_t mmr) {

    opcode = 0x02;
    sel = 0x00;
    op1 = gpr;
    op2 = mmr;

    transmit_instruction();

}
// example: ldr(r0, "[]", vga_colour)
void ldr(gpr_t gpr, string a, mmr_t deref_mmr) {

    opcode = 0x02;
    sel = 0x01;
    op1 = gpr;
    op2 = deref_mmr;

    transmit_instruction();

}
// example: ldr(r0, num1)
void ldr(gpr_t gpr, string var_name) {

    for(int i=0; i<2996; i++) {

        if(var_names[i] == var_name) {

            ldr_idx = i;
            break;
        }
    }

    // implicity calls ldr
    ldr(gpr,"[]",stack_addr[ldr_idx]);



}

// example: cmp(r0, 1)
void cmp(gpr_t gpr, uint16_t literal) {

    opcode = 0x04;
    sel = 0x02;
    op1 = gpr;
    op2 = literal;

    transmit_instruction();

}
// example: cmp(r0, r1)
```

```
346. void cmp(gpr_t gpr1, gpr_t gpr2) {
347.
348.     opcode = 0x04;
349.     sel = 0x03;
350.     op1 = gpr1;
351.     op2 = gpr2;
352.
353.     transmit_instruction();
354.
355. }
356.
357. // example: b(61)
358. void b(uint16_t addr) {
359.
360.     opcode = 0x05;
361.     sel = 0x00; // dc
362.     op1 = 0x00; // dc
363.     op2 = addr;
364.
365.     transmit_instruction();
366.
367. }
368.
369. // example: bgt(61)
370. void bgt(uint16_t addr) {
371.
372.     opcode = 0x06;
373.     sel = 0x00; // dc
374.     op1 = 0x00; // dc
375.     op2 = addr;
376.
377.     transmit_instruction();
378.
379. }
380.
381. // example: blt(61)
382. void blt(uint16_t addr) {
383.
384.     opcode = 0x07;
385.     sel = 0x00; // dc
386.     op1 = 0x00; // dc
387.     op2 = addr;
388.
389.     transmit_instruction();
390.
391. }
392.
393. // example: beq(61)
394. void beq(uint16_t addr) {
395.
396.     opcode = 0x08;
397.     sel = 0x00; // dc
398.     op1 = 0x00; // dc
399.     op2 = addr;
400.
401.     transmit_instruction();
402.
403. }
404.
405. // example: add(r2, 11,10)
406. void add(gpr_t gpr_result, uint16_t literal1, uint16_t literal2) {
407.
408.     opcode = 0x09;
409.     sel = 0x00;
410.     op1 = literal1;
411.     op2 = literal2;
412.
413.     op1h = (gpr_result >> 8) | (literal1 >> 8);
414.     op1l = op1 & 0x00FF;
415.     op2h = op2 >> 8;
416.     op2l = op2 & 0x00FF;
```

```
417.
418.     transmit_ALinstruction();
419.
420. }
421. // example: add(r2, 11,r0)
422. void add(gpr_t gpr_result, uint16_t literal1, gpr_t gpr_op2) {
423.
424.     opcode = 0x09;
425.     sel = 0x01;
426.     op1 = literal1;
427.     op2 = gpr_op2;
428.
429.     op1h = (gpr_result >> 8) | (literal1 >> 8);
430.     op1l = op1 & 0x00FF;
431.     op2h = op2 >> 12;
432.     op2l = 0x00;    // dc
433.
434.     transmit_ALinstruction();
435.
436. }
437. // example: add(r2, r1,10)
438. void add(gpr_t gpr_result, gpr_t gpr_op1, uint16_t literal2) {
439.
440.     opcode = 0x09;
441.     sel = 0x02;
442.     op1 = gpr_op1;
443.     op2 = literal2;
444.
445.     op1h = (gpr_result >> 8) | (gpr_op1 >> 12);
446.     op1l = op1 & 0x00FF;
447.     op2h = op2 >> 8;
448.     op2l = op2 & 0x00FF;
449.
450.     transmit_ALinstruction();
451.
452. }
453. // example: add(r2, r1,r1)
454. void add(gpr_t gpr_result, gpr_t gpr_op1, gpr_t gpr_op2) {
455.
456.     opcode = 0x09;
457.     sel = 0x03;
458.     op1 = gpr_op1;
459.     op2 = gpr_op2;
460.
461.     op1h = (gpr_result >> 8) | (gpr_op1 >> 12);
462.     op1l = op1 & 0x00FF;
463.     op2h = op2 >> 12;
464.     op2l = op2 & 0x00FF;
465.
466.     transmit_ALinstruction();
467.
468. }
469.
470. // example: sub(r2, 11,10)
471. void sub(gpr_t gpr_result, uint16_t literal1, uint16_t literal2) {
472.
473.     opcode = 0x0A;
474.     sel = 0x00;
475.     op1 = literal1;
476.     op2 = literal2;
477.
478.     op1h = (gpr_result >> 8) | (literal1 >> 8);
479.     op1l = op1 & 0x00FF;
480.     op2h = op2 >> 8;
481.     op2l = op2 & 0x00FF;
482.
483.     transmit_ALinstruction();
484.
485. }
486. // example: sub(r2, 11,r0)
487. void sub(gpr_t gpr_result, uint16_t literal1, gpr_t gpr_op2) {
```

```
488.
489.     opcode = 0x0A;
490.     sel = 0x01;
491.     op1 = literal1;
492.     op2 = gpr_op2;
493.
494.     op1h = (gpr_result >> 8) | (literal1 >> 8);
495.     op1l = op1 & 0x00FF;
496.     op2h = op2 >> 12;
497.     op2l = 0x00;    // dc
498.
499.     transmit_ALinstruction();
500.
501. }
502. // example: sub(r2, r1,10)
503. void sub(gpr_t gpr_result, gpr_t gpr_op1, uint16_t literal2) {
504.
505.     opcode = 0x0A;
506.     sel = 0x02;
507.     op1 = gpr_op1;
508.     op2 = literal2;
509.
510.     op1h = (gpr_result >> 8) | (gpr_op1 >> 12);
511.     op1l = op1 & 0x00FF;
512.     op2h = op2 >> 8;
513.     op2l = op2 & 0x00FF;
514.
515.     transmit_ALinstruction();
516.
517. }
518. // example: sub(r2, r1,r1)
519. void sub(gpr_t gpr_result, gpr_t gpr_op1, gpr_t gpr_op2) {
520.
521.     opcode = 0x0A;
522.     sel = 0x03;
523.     op1 = gpr_op1;
524.     op2 = gpr_op2;
525.
526.     op1h = (gpr_result >> 8) | (gpr_op1 >> 12);
527.     op1l = op1 & 0x00FF;
528.     op2h = op2 >> 12;
529.     op2l = op2 & 0x00FF;
530.
531.     transmit_ALinstruction();
532.
533. }
534.
535. // example: mul(r2, 11,10)
536. void mul(gpr_t gpr_result, uint16_t literal1, uint16_t literal2) {
537.
538.     opcode = 0x0B;
539.     sel = 0x00;
540.     op1 = literal1;
541.     op2 = literal2;
542.
543.     op1h = (gpr_result >> 8) | (literal1 >> 8);
544.     op1l = op1 & 0x00FF;
545.     op2h = op2 >> 8;
546.     op2l = op2 & 0x00FF;
547.
548.     transmit_ALinstruction();
549.
550. }
551. // example: mul(r2, 11,r0)
552. void mul(gpr_t gpr_result, uint16_t literal1, gpr_t gpr_op2) {
553.
554.     opcode = 0x0B;
555.     sel = 0x01;
556.     op1 = literal1;
557.     op2 = gpr_op2;
558.
```

```
559.    op1h = (gpr_result >> 8) | (literal1 >> 8);
560.    op1l = op1 & 0x00FF;
561.    op2h = op2 >> 12;
562.    op2l = 0x00;    // dc
563.
564.    transmit_ALinstruction();
565.
566. }
567. // example: mul(r2, r1,10)
568. void mul(gpr_t gpr_result, gpr_t gpr_op1, uint16_t literal2) {
569.
570.    opcode = 0x0B;
571.    sel = 0x02;
572.    op1 = gpr_op1;
573.    op2 = literal2;
574.
575.    op1h = (gpr_result >> 8) | (gpr_op1 >> 12);
576.    op1l = op1 & 0x00FF;
577.    op2h = op2 >> 8;
578.    op2l = op2 & 0x00FF;
579.
580.    transmit_ALinstruction();
581.
582. }
583. // example: mul(r2, r1,r1)
584. void mul(gpr_t gpr_result, gpr_t gpr_op1, gpr_t gpr_op2) {
585.
586.    opcode = 0x0B;
587.    sel = 0x03;
588.    op1 = gpr_op1;
589.    op2 = gpr_op2;
590.
591.    op1h = (gpr_result >> 8) | (gpr_op1 >> 12);
592.    op1l = op1 & 0x00FF;
593.    op2h = op2 >> 12;
594.    op2l = op2 & 0x00FF;
595.
596.    transmit_ALinstruction();
597.
598. }
599.
600. // example: lsr(r2, r1,1)
601. void lsr(gpr_t gpr_result, gpr_t gpr_op1, uint16_t literal2) {
602.
603.    opcode = 0x0C;
604.    sel = 0x02;
605.    op1 = gpr_op1;
606.    op2 = literal2;
607.
608.    op1h = (gpr_result >> 8) | (gpr_op1 >> 12);
609.    op1l = op1 & 0x00FF;
610.    op2h = op2 >> 8;
611.    op2l = op2 & 0x00FF;
612.
613.    transmit_ALinstruction();
614.
615. }
616.
617. // example: AND(r2, 11,10)
618. void AND(gpr_t gpr_result, uint16_t literal1, uint16_t literal2) {
619.
620.    opcode = 0x0D;
621.    sel = 0x00;
622.    op1 = literal1;
623.    op2 = literal2;
624.
625.    op1h = (gpr_result >> 8) | (literal1 >> 8);
626.    op1l = op1 & 0x00FF;
627.    op2h = op2 >> 8;
628.    op2l = op2 & 0x00FF;
629.
```

```
630.     transmit_ALinstruction();
631.
632. }
633. // example: AND(r2, 11,r0)
634. void AND(gpr_t gpr_result, uint16_t literal1, gpr_t gpr_op2) {
635.
636.     opcode = 0x0D;
637.     sel = 0x01;
638.     op1 = literal1;
639.     op2 = gpr_op2;
640.
641.     op1h = (gpr_result >> 8) | (literal1 >> 8);
642.     op1l = op1 & 0x00FF;
643.     op2h = op2 >> 12;
644.     op2l = 0x00;    // dc
645.
646.     transmit_ALinstruction();
647.
648. }
649. // example: AND(r2, r1,10)
650. void AND(gpr_t gpr_result, gpr_t gpr_op1, uint16_t literal2) {
651.
652.     opcode = 0x0D;
653.     sel = 0x02;
654.     op1 = gpr_op1;
655.     op2 = literal2;
656.
657.     op1h = (gpr_result >> 8) | (gpr_op1 >> 12);
658.     op1l = op1 & 0x00FF;
659.     op2h = op2 >> 8;
660.     op2l = op2 & 0x00FF;
661.
662.     transmit_ALinstruction();
663.
664. }
665. // example: AND(r2, r1,r1)
666. void AND(gpr_t gpr_result, gpr_t gpr_op1, gpr_t gpr_op2) {
667.
668.     opcode = 0x0D;
669.     sel = 0x03;
670.     op1 = gpr_op1;
671.     op2 = gpr_op2;
672.
673.     op1h = (gpr_result >> 8) | (gpr_op1 >> 12);
674.     op1l = op1 & 0x00FF;
675.     op2h = op2 >> 12;
676.     op2l = op2 & 0x00FF;
677.
678.     transmit_ALinstruction();
679.
680. }
681.
682. // example: OR(r2, 11,10)
683. void OR(gpr_t gpr_result, uint16_t literal1, uint16_t literal2) {
684.
685.     opcode = 0x0E;
686.     sel = 0x00;
687.     op1 = literal1;
688.     op2 = literal2;
689.
690.     op1h = (gpr_result >> 8) | (literal1 >> 8);
691.     op1l = op1 & 0x00FF;
692.     op2h = op2 >> 8;
693.     op2l = op2 & 0x00FF;
694.
695.     transmit_ALinstruction();
696.
697. }
698. // example: OR(r2, 11,r0)
699. void OR(gpr_t gpr_result, uint16_t literal1, gpr_t gpr_op2) {
700.
```

```
701.    opcode = 0x0E;
702.    sel = 0x01;
703.    op1 = literal1;
704.    op2 = gpr_op2;
705.
706.    op1h = (gpr_result >> 8) | (literal1 >> 8);
707.    op1l = op1 & 0x00FF;
708.    op2h = op2 >> 12;
709.    op2l = 0x00;    // dc
710.
711.    transmit_ALinstruction();
712.
713. }
714. // example: OR(r2, r1,10)
715. void OR(gpr_t gpr_result, gpr_t gpr_op1, uint16_t literal2) {
716.
717.    opcode = 0x0E;
718.    sel = 0x02;
719.    op1 = gpr_op1;
720.    op2 = literal2;
721.
722.    op1h = (gpr_result >> 8) | (gpr_op1 >> 12);
723.    op1l = op1 & 0x00FF;
724.    op2h = op2 >> 8;
725.    op2l = op2 & 0x00FF;
726.
727.    transmit_ALinstruction();
728.
729. }
730. // example: OR(r2, r1,r1)
731. void OR(gpr_t gpr_result, gpr_t gpr_op1, gpr_t gpr_op2) {
732.
733.    opcode = 0x0E;
734.    sel = 0x03;
735.    op1 = gpr_op1;
736.    op2 = gpr_op2;
737.
738.    op1h = (gpr_result >> 8) | (gpr_op1 >> 12);
739.    op1l = op1 & 0x00FF;
740.    op2h = op2 >> 12;
741.    op2l = op2 & 0x00FF;
742.
743.    transmit_ALinstruction();
744.
745. }
746.
747. // example: mvn(r0, 0)
748. void mvn(gpr_t gpr, uint16_t literal) {
749.
750.    opcode = 0x0F;
751.    sel = 0x00;
752.    op1 = gpr;
753.    op2 = literal;
754.
755.    transmit_instruction();
756.
757. }
758. // example: mvn(r2, r1)
759. void mvn(gpr_t gpr1, gpr_t gpr2) {
760.
761.    opcode = 0x0F;
762.    sel = 0x01;
763.    op1 = gpr1;
764.    op2 = gpr2;
765.
766.    transmit_instruction();
767.
768. }
769.
770.
```

A2 Example Assembly Programs

A2.1 Blinky

```
1. str(0x00, led); // LEDs off initially
2. str(100, tmr); // start 100ms tmr
3. ldr(r0, "[]", tmr); // timer state
4. cmp(r0, 0x01);  // if done
5. beq(6); // if
6. b(2); // else
7. str(0xFF, led); // LEDs on
8. str(0x00, tmr); // stop tmr;
9. str(100, tmr); // start 100ms timer
10. ldr(r0, "[]", tmr); // timer state
11. cmp(r0, 0x01);  // if done
12. beq(13); // if
13. b(9); // else
14. str(0x00, led); // LEDs off
15. str(0x00, tmr); // stop tmr;
16. b(1);
17.
```

A2.2 Piano

```
1. str(0x01, vga_sel); // manual rendering mode
2. ldr(r0, "[]", keyboard_input); // read key pressed
3. cmp(r0, 0x1C); // A
4. beq(5); // if
5. b(13); // else
6. str(0x1C, audio); // A note
7. str(0xF00, vga_colour); // red
8. ldr(r0, "[]", keyboard_input); // read key pressed
9. cmp(r0, 0x00); // key released
10. beq(11); // if
11. b(7); // else
12. str(0x00, audio); // tune off
13. str(0x00, vga_colour); // display off
14. cmp(r0, 0x32); // B
15. beq(16); // if
16. b(24); // else
17. str(0x32, audio); // B note
18. str(0xE80, vga_colour); // orange
19. ldr(r0, "[]", keyboard_input); // read key pressed
20. cmp(r0, 0x00); // key released
21. beq(22); // if
22. b(18); // else
23. str(0x00, audio); // tune off
24. str(0x00, vga_colour); // display off
25. cmp(r0, 0x21); // C
26. beq(27); // if
27. b(35); // else
28. str(0x21, audio); // C note
29. str(0xFF0, vga_colour); // yellow
30. ldr(r0, "[]", keyboard_input); // read key pressed
31. cmp(r0, 0x00); // key released
32. beq(33); // if
33. b(29); // else
34. str(0x00, audio); // tune off
35. str(0x00, vga_colour); // display off
```

```
36. cmp(r0, 0x23); // D
37. beq(38); // if
38. b(46); // else
39. str(0x23, audio); // D note
40. str(0x0F0, vga_colour); // green
41. ldr(r0, "[]", keyboard_input); // read key pressed
42. cmp(r0, 0x00); // key released
43. beq(44); // if
44. b(40); // else
45. str(0x00, audio); // tune off
46. str(0x00, vga_colour); // display off
47. cmp(r0, 0x24); // E
48. beq(49); // if
49. b(57); // else
50. str(0x24, audio); // E note
51. str(0x00F, vga_colour); // blue
52. ldr(r0, "[]", keyboard_input); // read key pressed
53. cmp(r0, 0x00); // key released
54. beq(55); // if
55. b(51); // else
56. str(0x00, audio); // tune off
57. str(0x00, vga_colour); // display off
58. cmp(r0, 0x2B); // F
59. beq(60); // if
60. b(68); // else
61. str(0x2B, audio); // F note
62. str(0xB3E, vga_colour); // indigo
63. ldr(r0, "[]", keyboard_input); // read key pressed
64. cmp(r0, 0x00); // key released
65. beq(66); // if
66. b(62); // else
67. str(0x00, audio); // tune off
68. str(0x00, vga_colour); // display off
69. cmp(r0, 0x34); // G
70. beq(71); // if
71. b(1); // else
72. str(0x34, audio); // G note
73. str(0x83F, vga_colour); // violet
74. ldr(r0, "[]", keyboard_input); // read key pressed
75. cmp(r0, 0x00); // key released
76. beq(77); // if
77. b(73); // else
78. str(0x00, audio); // tune off
79. str(0x00, vga_colour); // display off
80. b(1);
```

A2.3 VGA Classic RGB Pattern

```
1. str(0x01, vga_sel); // manual rendering mode
2. mov(r1, 0xF00); // red
3. mov(r2, 0x0F0); // green
4. mov(r3, 0x00F); // blue
5. ldr(r0, "[]", vga_column);
6. cmp(r0, 213); // if column < 213
7. blt(11); // if
8. cmp(r0, 426); // if column < 426
9. blt(13); // if
```

```
10. cmp(r0, 639); // if column < 639
11. blt(15); // if
12. str(r1, vga_colour);
13. b(4);
14. str(r2, vga_colour);
15. b(4);
16. str(r3, vga_colour);
17. b(4);
```

A3 CPU Hardware Timers

There can be as many hardware timers connected to memory as desired. It is a memory mapped peripheral and can be interacted with using assembly instructions. **This timer has a maximum delay of ~5.36 seconds.**

*To **start** the timer:*

```
1. str(T, tmr); // start Tms tmr
```

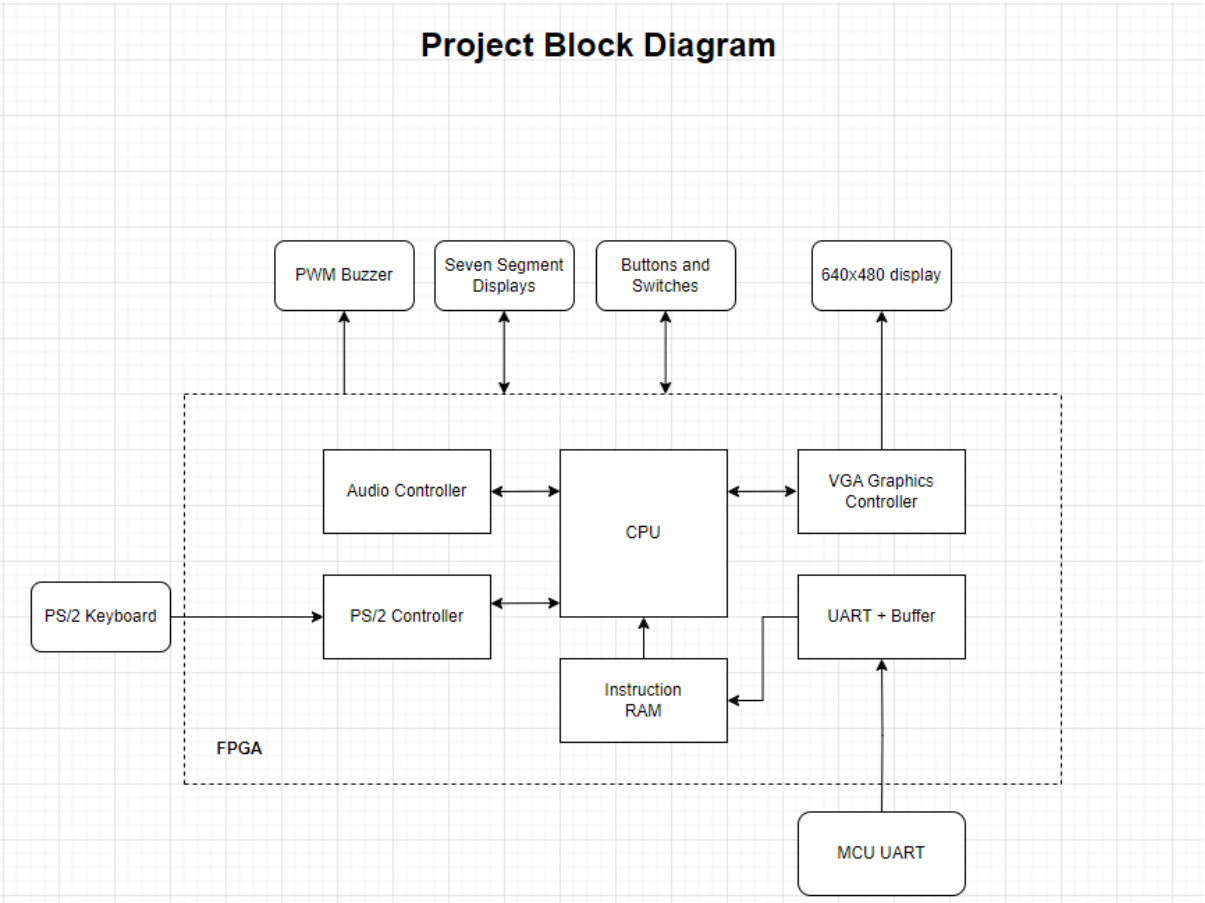
*To **stop** the timer:*

```
1. str(0x00, tmr); // stop tmr;
```

As this is such an important part of the CPU, the full RTL code is shown below.

```
1.  module CPUTimer (
2.
3.    output logic rdy,
4.
5.    input logic [12:0] Tms,
6.    input logic clk, rst
7.  );
8.
9.  ////// calculations based off 50MHz clock //////
10.
11. // max value is 4294 million (4294 million seconds max delay)
12. logic [27:0] count_reg = 'd0;
13.
14. // 28-bits: max delay ~5.36s
15. logic [27:0] N;
16.
17. // N is not static, so use an always comb block
18. // not logic [27:0] N = ...
19. // or parameter N = ...
20. // as both of the above are static!
21. always_comb begin
22.
23.   // Tms is the required delay, in ms
24.   N = Tms*50000;
25.
26. end
27.
28. always_ff @(posedge clk) begin
29.
30. if(rst) begin
31.   rdy <= 0;
32.   count_reg <= 'd0;
33.
34. end
35.
36. else begin
37.
38.   // timer stop
39.   if(Tms == 'd0) begin
40.
41.   rdy <= 0;
42.   count_reg <= 'd0;
43.
44. end
45.
46. // timer start
47. else begin
48.
49.   if(count_reg != N) begin
50.
51.   count_reg <= count_reg + 1;
52.   // not rdy <= 0; here
53.   // the CPU may miss it for whatever reason
54.   // instead, have it latch until the CPU stops the timer
55.
56. end
57.
58. else begin
59.   rdy <= 1;
60.   count_reg <= 'd0;   // doesn't matter as CPU will stop and reset timer by sending 0
    anyways
61. end
62.
63. end
64.
65. end
66.
67. end
68.
69. endmodule
```

A4 Project Block Diagram



A5 Project Warnings

1. Data transmission may be corrupted if an object that possesses an electric/magnetic field is placed near the Tx wire. This is possibly due to the E/H fields interacting with the E field around the Tx wire, changing its voltage level.

2. A ground loop in electronics occurs when there are multiple paths to ground, creating unintended current flows that can cause noise or interference. When connecting isolated power supplies (PSUs), ground loops can occur if the grounds of the supplies are connected, leading to potential issues like signal degradation or electrical hazards. To avoid this, it's important to maintain isolation between the grounds of each power supply. *[chat.openai.com].* This project uses a 5V PSU that powers the LED panel that is isolated from the 5V USB powered FPGA and MCU. A ground loop may occur in which case the UART transmission may be corrupted. Isolation techniques like using optocouplers (aka optoisolators) may be used.

A6 Project Evaluation

Health and Safety	During project design: eye strain; back pain; ergonomics due to long coding sessions. Running of Project: • LED panel can draw upwards of 5A, a PSU with current limiting capabilities must be used for safety. • LEDs can get very bright and flash at high rates – risk of photosensitive epilepsy.
Legal Requirements	None
Intellectual Property	Intel IP cores may be used
Environmental Impact	None; project does not harm the environment
Ethical Conduct	Project does not harm other people