FH JOANNEUM
Elektronik und
Computer Engineering

MIC-1

ECE20 Project

WS22

Claudia Lux & Valentina Ceru & Emile Schockaert & Faiz Sehic & Andreas Greiner
4. März 2023

Inhaltsverzeichnis

1 Hardware to exchange programms

By generating the bitstream of the top-level design a program will be stored in the Main-Memory of the MIC-1. This program can be defined inside *SystemVerilog/defines_ add.sv*.
In order to exchange to program which is stored inside the Main-Memory a Memory-Manager, UART-Register and a additional Multiplexer was designed.

1.1 Memory-Manager

The purpose of the Memory-Manager is to switch the connection of the main-memory from the MIC-1 to the UART-Register.

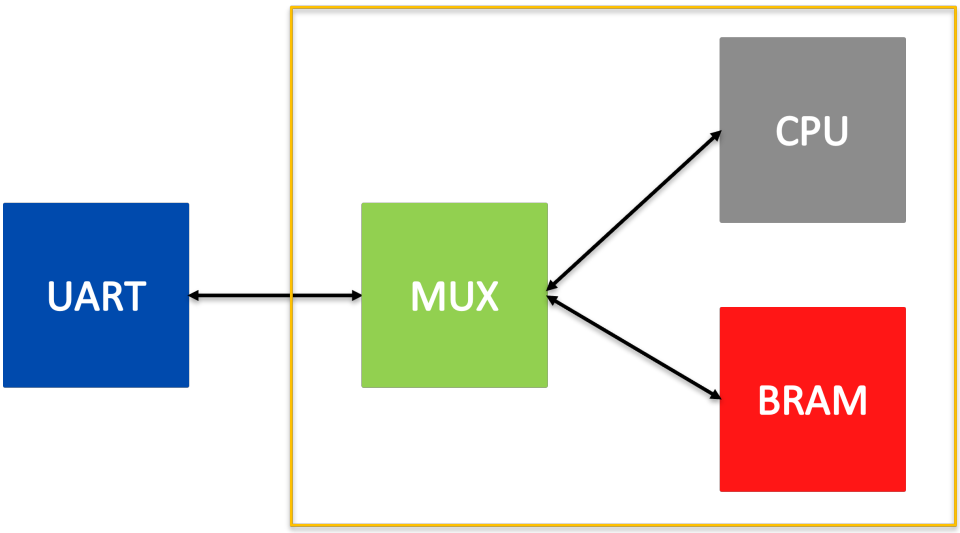

Abbildung 1.1: Blockdiagram of the new connection

1.1.1 Task

First it was needed to create a multipexer, the corresponding code can be seen in the code section. This module was later than integrated in the mic1_soc". For the integration of the module we needed to break up the old connections from the main-memoryto the mic1". The new inputs from the Uart and the outputs of the mic1"were assigned as inputs of the memory-manager and the outputs of the memory-manager assigned input of the main-memory. By this we can switch the inputs of the main-memory"between the mic1 and uart and overwrite the existing program.

1.1.2 Schematic

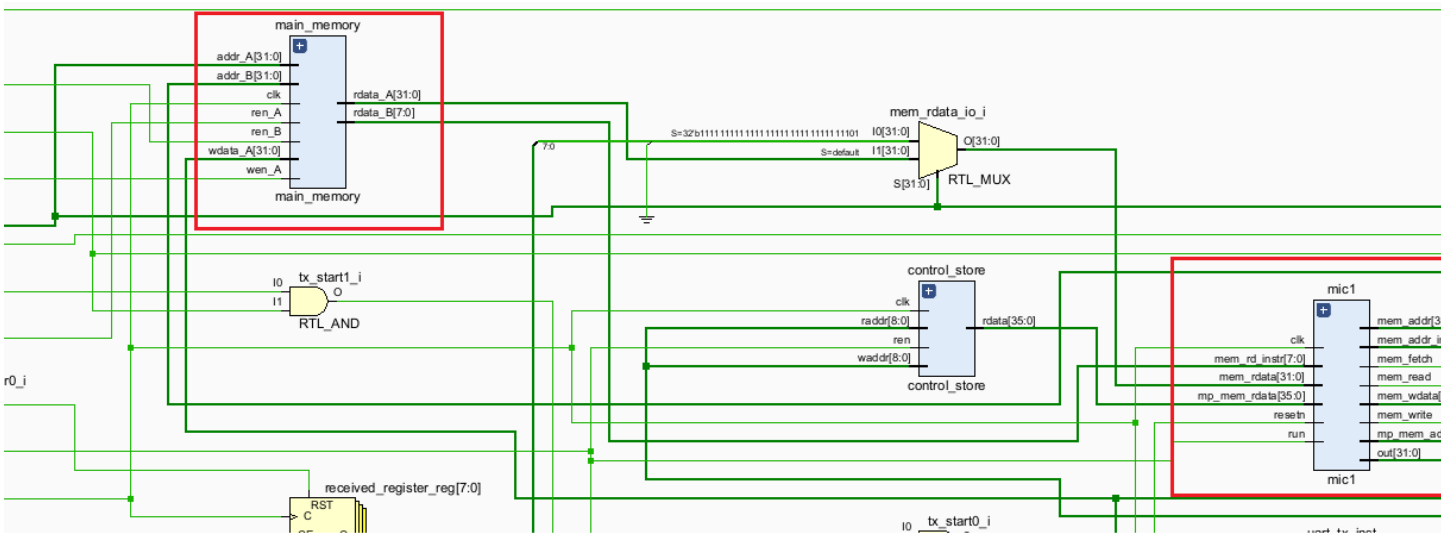

Abbildung 1.2: Schematic of the old connection

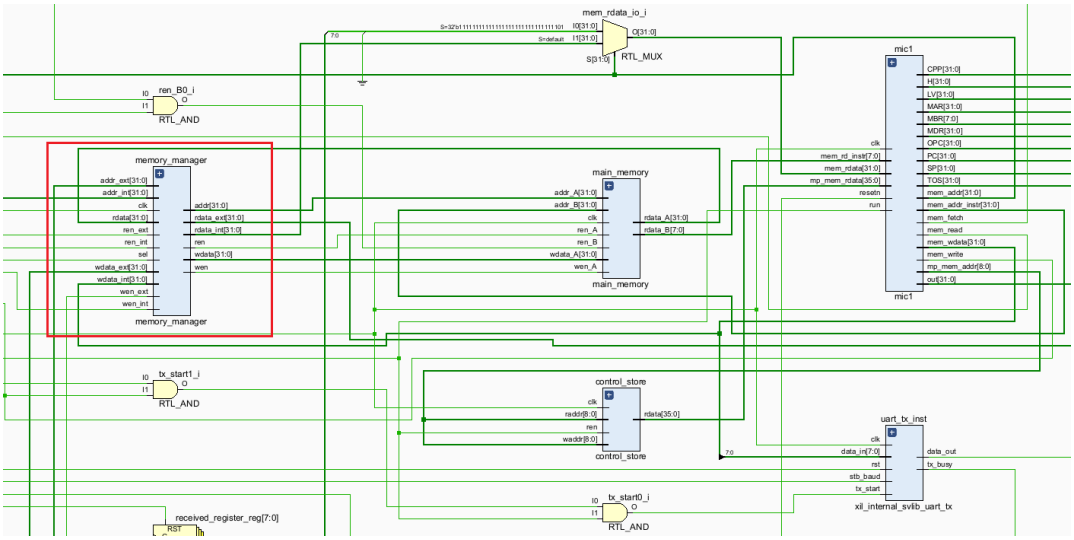

Abbildung 1.3: Schematic of the new connection

1.1.3 Code

Listing 1.1: Memory-Manager.

```
module memory_manager(
    input logic clk, sel,
    input logic wen_int, ren_int,
    input logic wen_ext, ren_ext,
    input logic [31:0] addr_int, wdata_int,
    input logic [31:0] addr_ext, wdata_ext,
    input logic [31:0] rdata,
    output logic wen, ren,
    output logic [31:0] addr, wdata, rdata_int, rdata_ext
    );

    always_comb begin
        case (sel)
        1'b0:
            begin
                wen = wen_int;
                ren = ren_int;
                addr = addr_int;
                wdata = wdata_int;
                rdata_int = rdata;
                rdata_ext =  'h00000000;
            end
        1'b1:
            begin
                wen = wen_ext;
                ren = ren_ext;
                addr = addr_ext;
                wdata = wdata_ext;
                rdata_int = 'h00000000;
                rdata_ext =  rdata;
            end
        endcase
    end
endmodule
```

Listing 1.2: Integration of the Memory-Manager in the MIC-1 SOC.

```
    memory_manager#(
    )memory_manager (
    .clk        (clk),
    .sel        (sel_uart),
    .wen_int    (mem_write && mic1_run),
    .ren_int    (mem_read  && mic1_run),
    .addr_int   (mem_addr),
    .wdata_int  (mem_wdata),
    .wen_ext    (w_en),
    .ren_ext    (r_en),
    .addr_ext   (w_adress),
    .wdata_ext  (w_data),
    .rdata_int  (mem_rdata),
    .rdata_ext  (r_data),
    .wen        (mm_wen),
    .ren        (mm_ren),
    .addr       (mm_addr),
    .wdata      (mm_wdata),
    .rdata      (mm_rdata)
    );
endmodule
```

1.2 UART module and multiplexing the UART interfaces

1.2.1 Task

The already existing implementation of the MIC-1 processor, shall be extended with a UART-Interface, which enables to write data directly into the the registers of its BRAM. In the beginning the UART IP-Core was tested with a Virtual I/O block, to get familiar with its functionality. From this point the Interface was successively extended with a BRAM Block, to simulate the memory of the microprocessor. At this stage the implementation was ready to be tested with the GUI's interface. In the last process the virtual components were removed and connected with the existing top level design. A system analyzer could show the output at this moment, which was eventually replaced with the VGA Interface.
Previously the program to be run on the microprocessor was generated within the Bitstream. With the addition of the UART Module a new microprogramm can be sent to the Block Ram. Since the input data to the MBR (Memory Buffer register) and the UART Module use the same UART interface a multiplexer has to be added, to avoid collision of data. This multiplexer derives whether data shall be sent to the memory or straight to the MIC's ALU via the MBR and allows the communication to strictly either and keeps the other on IDLE (HIGH = "d1"). The switch is mapped to a hardware switch on the Basys3 board (T1) and well as the TX and RX are mapped to hardware ports (TX = B18, RX = A18). The clock was connected to the synchronized system clock.

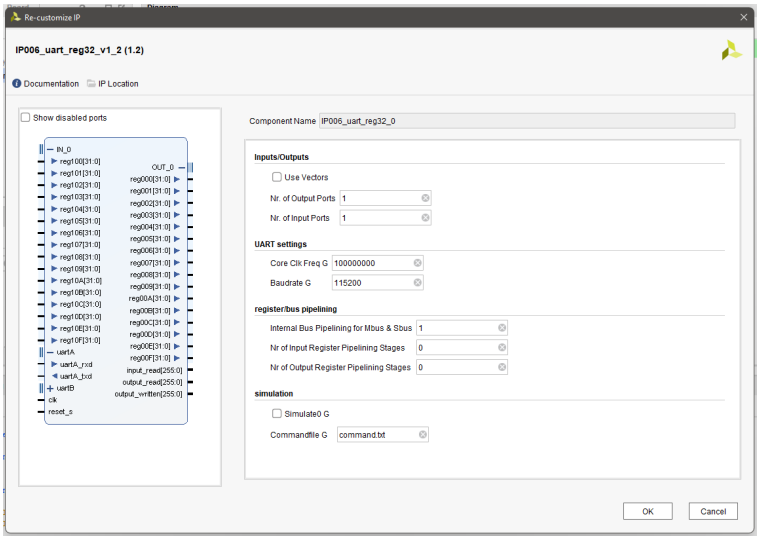

Abbildung 1.4: Adressing of the UART module

1.2.2 Schematic / Block-design

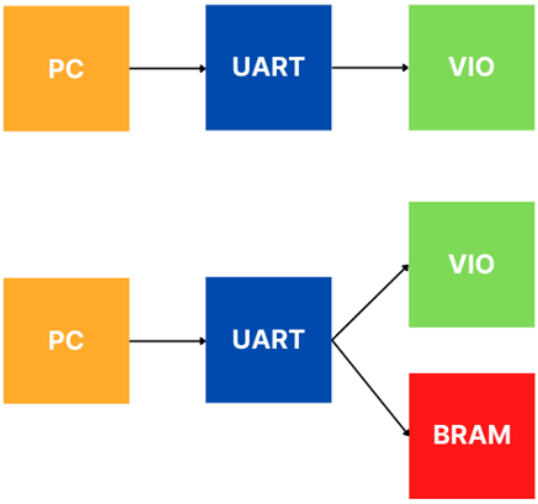

Abbildung 1.5: Blockdiagram of the two first testing stages

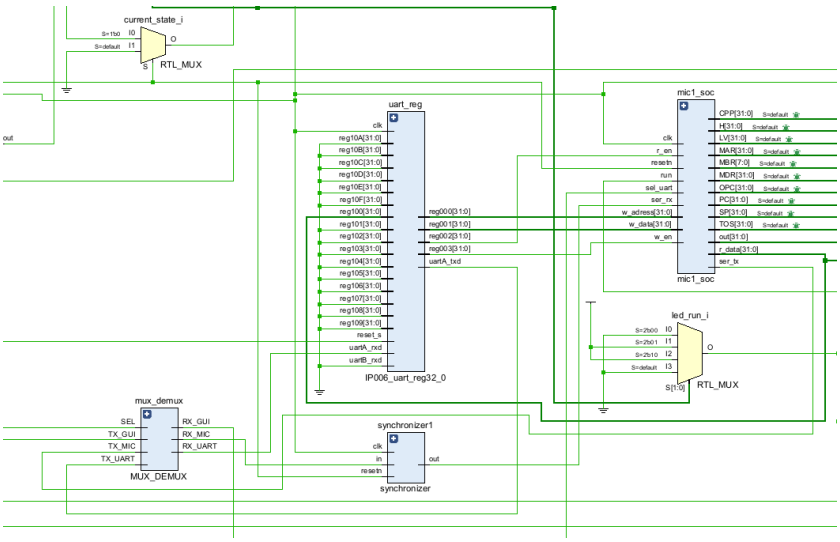

Abbildung 1.6: Schematic of the UART module including the MUX connected to the MIC SOC

All the tests were performed in the Block Design domain. The final implementation was done manually in VHDL.

1.2.3 Code

Listing 1.3: Source Code of the mux$_demux$.

```
`timescale 1ns / 1ps
module MUX_DEMUX(
    input SEL,
    input TX_GUI,
    input TX_UART,
    input TX_MIC,
    output RX_UART,
    output RX_MIC,
    output RX_GUI
    );
    reg RX_UART, RX_MIC, RX_GUI;
    always@*
    begin
        case (SEL)
            0 : begin
            RX_UART = TX_GUI;
            RX_GUI = TX_UART;
            RX_MIC = 1;
            end

            1 : begin
            RX_MIC = TX_GUI;
            RX_GUI = TX_MIC;
            RX_UART = 1;
            end
        endcase
    end
endmodule
```

1.3 Sending Data via UART (Data Syntax)

The UART module has four 32-bit output registers (reg000-003) which communicate with the BRAM. The first register contains the address of the memory cell, the second the data, reg002 enables writing while reg003 toggles the GLOBAL ENABLE. To address the UART output register the command to be sent looks like following:

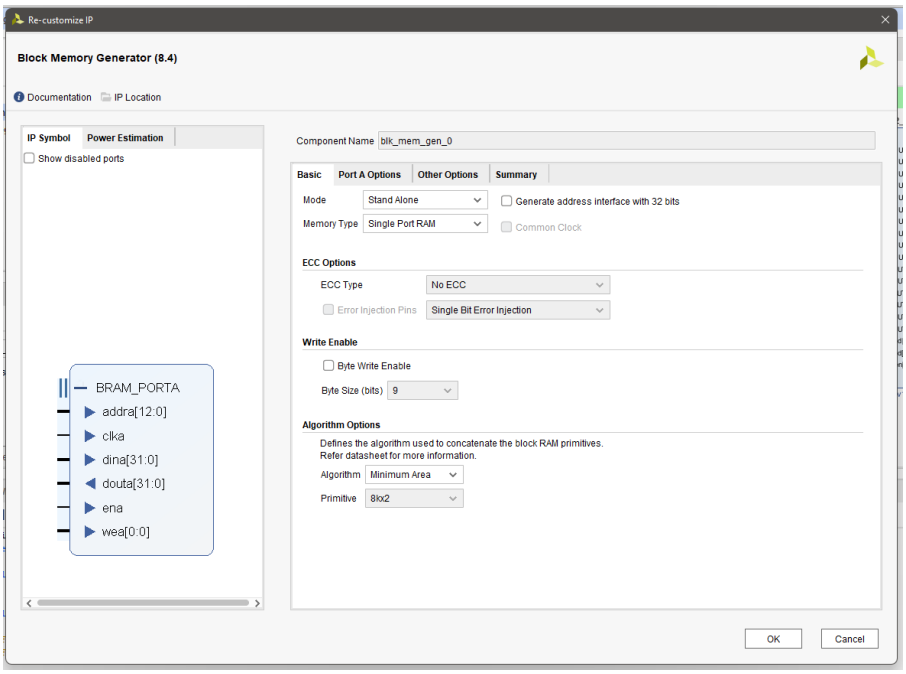

Abbildung 1.7: BRAM adressing

Abbildung 1.8: Syntax to send data to the UART output registers

with the first character reading (r) or writing (w) will be selected. The next 4 digit hex value addresses the output register, while the final 8 digits contain the actual data in hexadecimal.
and example to send 0xFFFF to address 0 of the BRAM:

w0000 0000 0000 -> select address 0
w0001 0000 FFFF -> write data 0xFFFF
w0002 0000 0001 -> enable the BRA
w0003 0000 0001 -> write enable

Name	Value	Activity	Direction	VIO
UART_GUI_i/blk_mem_gen_0_douta[31:0]	[H] 0000_FFFF		Input	hw_vio_1
UART_GUI_i/IP006_uart_reg32_0_reg000[31:0]	[H] 0000_0001		Input	hw_vio_1
UART_GUI_i/IP006_uart_reg32_0_reg001[31:0]	[H] 0000_FFFF		Input	hw_vio_1
UART_GUI_i/IP006_uart_reg32_0_reg002[31:0]	[H] 0000_0001		Input	hw_vio_1
UART_GUI_i/IP006_uart_reg32_0_reg003[31:0]	[H] 0000_0001		Input	hw_vio_1
UART_GUI_i/probe_in5_0_1[15:0]	[H] FFFF		Input	hw_vio_1

Abbildung 1.9: output at

2 VGA

2.1 Task

The purpose of the VGA controller will be to display the important registers of the MIC-1 microcontroller on the screen. These registers are the Stack Pointer, the Accumulator, the Program Counter and a few more. Therefore a VGA Controller has to be implemented in Vivado for the Basys3 FPGA board, which will be connected with the display over a VGA cable. The Controller will be implemented by using Verilog modules in Vivado and connecting them in a Block Design. In this implementation, a resolution of 640 pixels by 480 pixels is used for the screen, a refresh rate of 60Hz and 12 bits per pixel for the colors(red, green and blue). This part of the project will be an additional feature on an existing VGA Controller that one of the contributors of this project already worked on.

2.2 Original Design

In Figure 2.1 the original Block Design is shown. This design will be changed to fulfill the desired requirements, stated here above.
This implementation of the VGA Controller is a simple text editor. The inputs of the Block Design iIncr, iExtra, iStop are buttons and iSw0, iSw1 are sliders. Each slider could be on or off and for each position of the sliders the buttons have a different purpose. The implementation of the buttons could show a lower case character on the screen or make a space, enter, change the characters to upper case... The iRst is a button to reset the whole screen and the iClk is the input clock which is used for updating the screen. It is initialized at 100MHz but the Clocking Wizard module in Figure 2.1 decrements it to 25MHz. This clock signal ensures that we have a refresh rate of 60Hz.

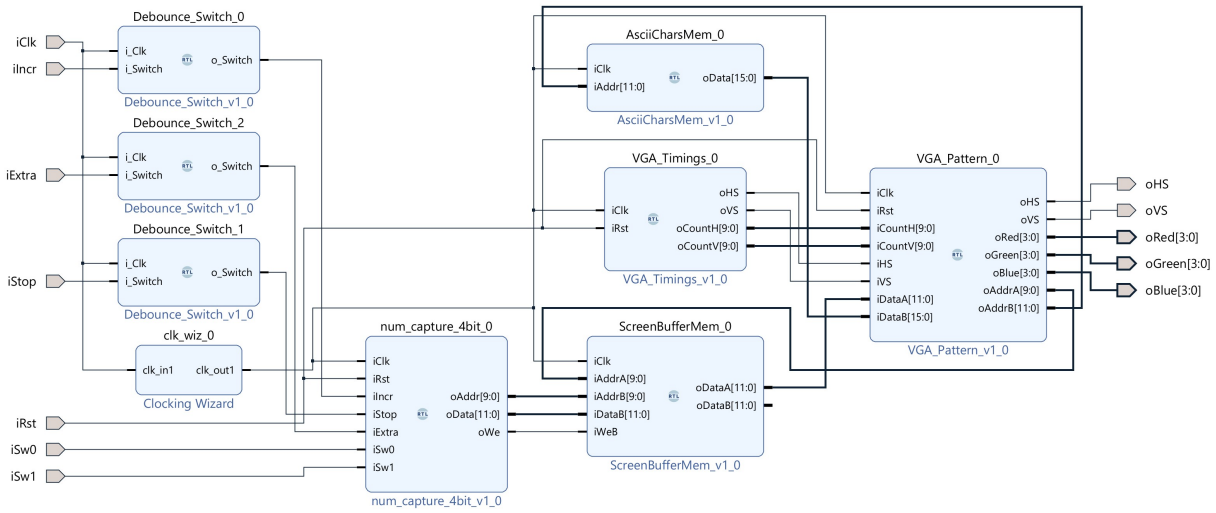

Abbildung 2.1: Original Block Design

The outputs of the block design are directly connected with the VGA module on the Basys3 board. The output ports oHS and oVS are respectively the horizontal sync and the vertical sync. This means that those represent the position which the VGA is updating the color of a pixel on the screen. The oRed, oGreen and oBlue are each 4 bit signal. Those are used to set the color of the pixel when the VGA is updating it.
The modules that are relevant for this project are the Debounce_Switch, the num_capture_4bit and the ScreenBufferMem. The num_capture_4bit is the module where the State Machine is defined and is discussed in

section 2.2.1. The ScreenBufferMem module is important to show the a Static part on the screen, this is also discussed further in the document in section 2.3.2. The Debounce_Switch module is used because when the switch is switched there is a small time when the slider is in an undefined state. This is unwanted, because if a button is pressed and the switch is undefined, there is no case that is valid, so it will give an error. In this case, this is because 0 and 1. It is shown in Figure 2.2. Here in the figure, 5 Volts corresponds that the Switch is ON, at 0 Volts the switch is OFF. For the buttons a Debounce module is not needed.

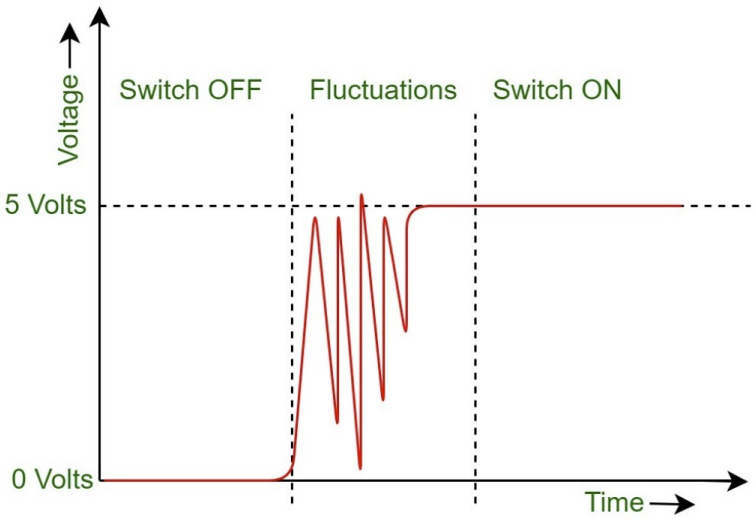

Abbildung 2.2: Debounce Switch

2.2.1 State Machine

This initial State Machine will start updating the screen in the top left pixel of the screen. Then it will go from left to right on the screen and when it reaches the right side, the row variable will be incremented so that the next pixel, is the pixel on the second row, first column. This will go on till it reaches the end of the screen and then the position will be reset to the initial position, the left top pixel.
The code of the State Machine is divided in four parts. The first one is the input, outputs and all internal variables and signals A.1. The next is the Next State Logic A.2. Then the Output Logic A.3 and the last one is the State Register A.4.

2.3 New Block Design

The new Block Design is shown in Figure 2.3. The difference from the original Block Design 2.1 is that there are a number of new inputs. Only the implementation of the reset button (iRst) and the clock (iClk) are kept the same. There is also an extra input switch (iSwUpdate), when the switch is ON the screen is updated otherwise the screen will keep on displaying the same values. Another additional output is added, this is the clock signal that is 25MHz (oClk25). This is added because this whole Block Design is added to the MIC1-project in Vivado and to have no complications it needs to work on the same clock signal. This means that the oClk25 is the input clock signal of the MIC1-project. Next the internal code of the num_capture_4bit module is changed, because there is the State Machine 2.3.1 defined.

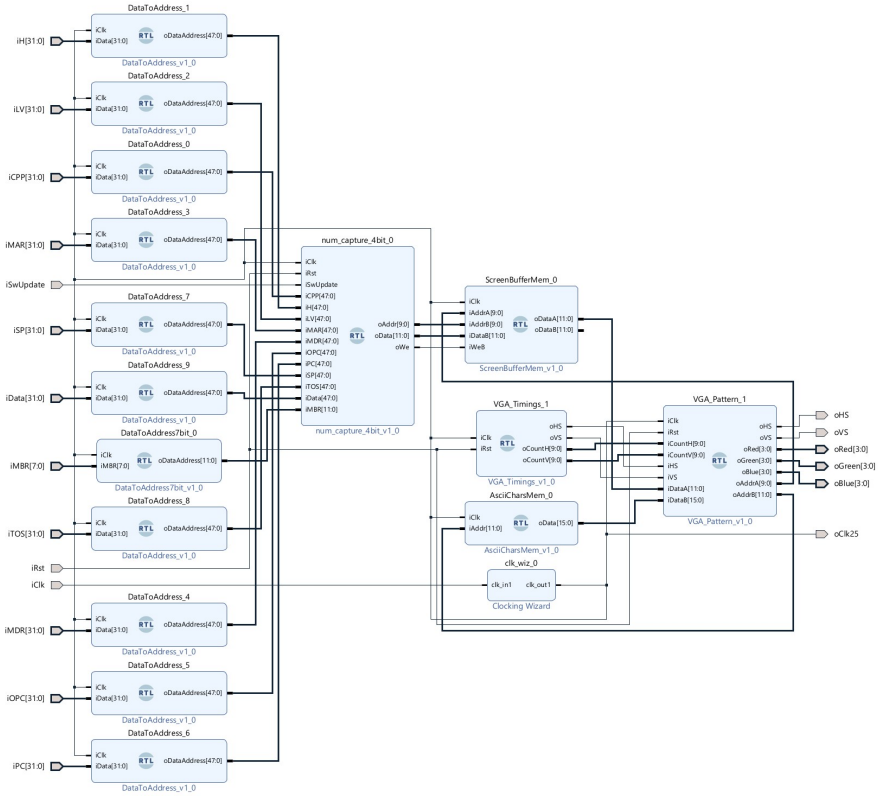

Abbildung 2.3: New Block Design

2.3.1 State Machine

Firstly all the input variables were added A.5, line 5-10. Also some states were deleted and update states were added, line 21-35. In these update states, the value of the internal register is continuously updated. For these new states the State Logic is defined A.6. For every case in the State Logic, there is defined when the current state needs to be updated. In the Output Logic A.6, the next pixel that needs to be updated is defined. Lastly in the State Register A.8 the value of the pixel is defined.

2.3.2 Static Screen

The purpose of the VGA Module is that it can show the internal registers of the MIC1 microcontroller. Thus on the screen will be the names of the registers and those values. The values need to be updateted all the time but the names are static. In the ScreenBufferMem module from Figure 2.3 a memory file can be uploaded that represents the static screen. Beware that the screen is divided in blocks to display the character. The screen has a size of 15x40 blocks.
If you want to change the static screen, follow these steps:

1. Make a .txt file with 15 rows and 40 columns that represent your new static screen, example 2.4. (Notepad++ recommended)
2. Save the .txt file on your computer.
3. Run this Python script A.9, it will open a file explorer where you need to select the .txt file.
4. The .mem file needed for the static screen gets saved in your Downloads folder.
5. Add the .mem file in the Vivado project and change the file in the ScreenBufferMem module to the new .mem file on line 21 in the code A.10.

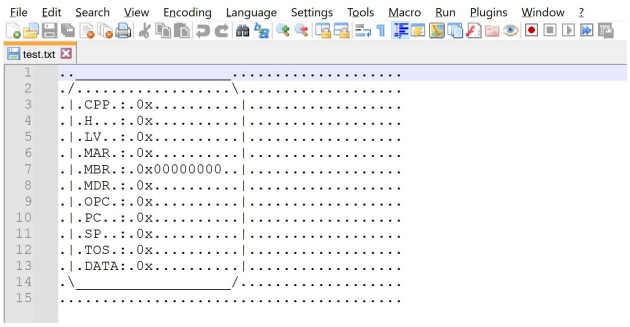

Abbildung 2.4: Example of a static screen in Notepad++

3 GUI

3.0.1 GUI

A Python file which includes a graphical user interface (GUI) was created to exchange the program on the MIC. With this GUI, it is easy and intuitive to interact with the MIC by connecting the Basys 3 board with a serial port. The TKinter framework was chosen as it is already included in the Python standard library and therefore no additional dependencies are required. TKinter is also very user- friendly.

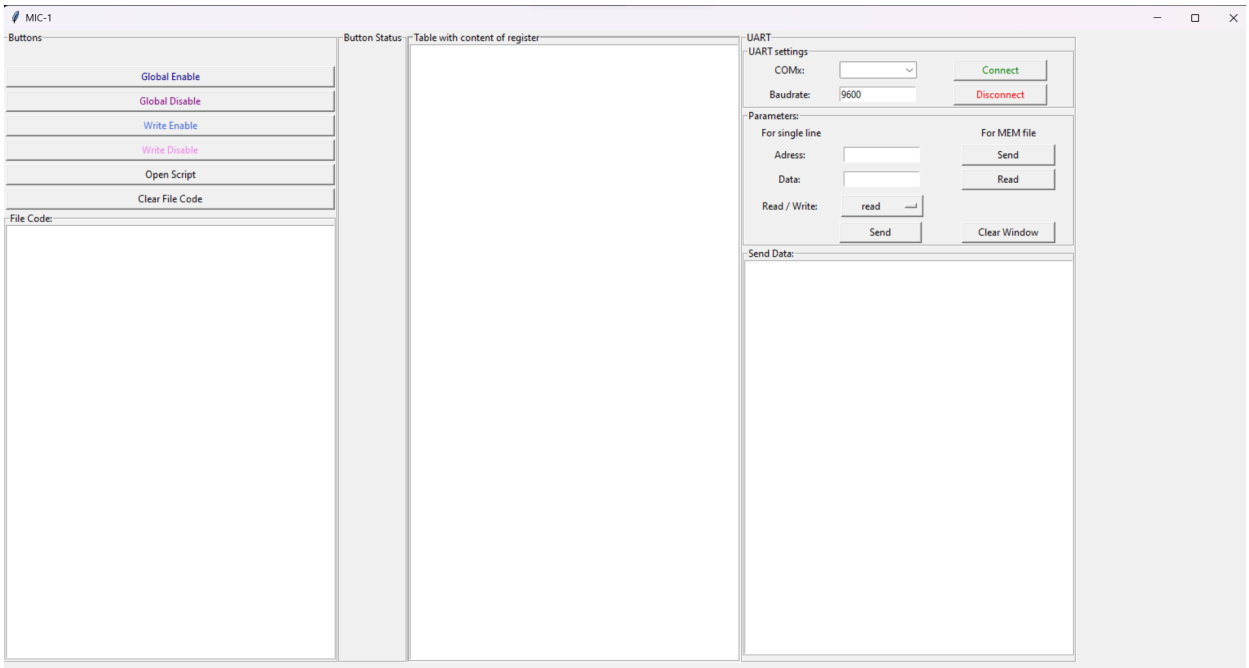

Abbildung 3.1: GUI

The GUI includes various Buttons. The upper four buttons are used when sending a single line to the MIC-1.

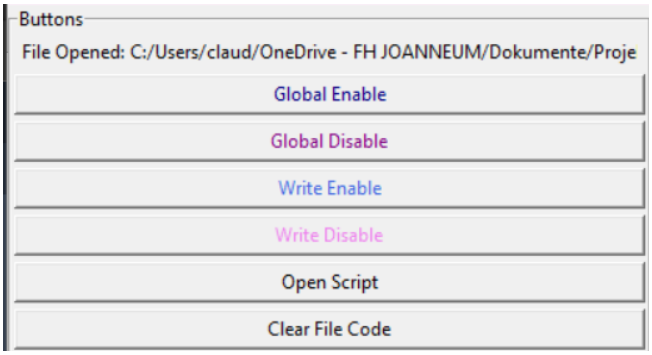

Abbildung 3.2: Buttons of the GUI

3.0.2 Protocol

Before sending a file, the MIC-1 should be connected via UART. To do that choose the COM-Port in the Window *COMx:*

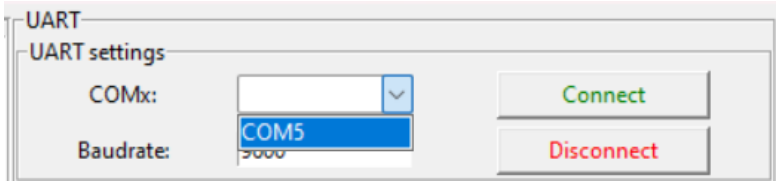

Abbildung 3.3: UART Connection

When the MIC-1 is successfully connected, the word Connect is displayed in the button table.

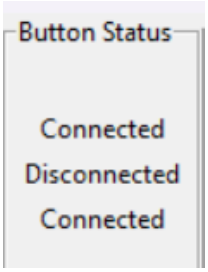

Abbildung 3.4: Completed UART Connection

The File path is displayed in the upper left corner of the GUI.The number of lines in the file is shown in the Button Status Table. In addition, the content of the file is displayed in the *File Code* Window.

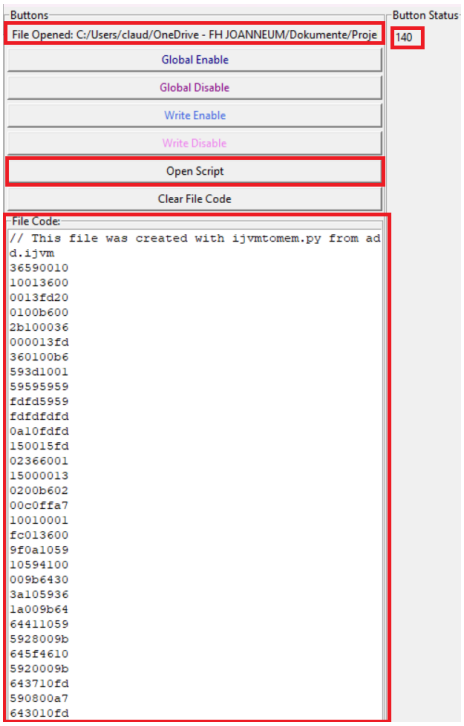

Abbildung 3.5: Open a mem file

The mem file can be selected after pressing the button *Open Script*.

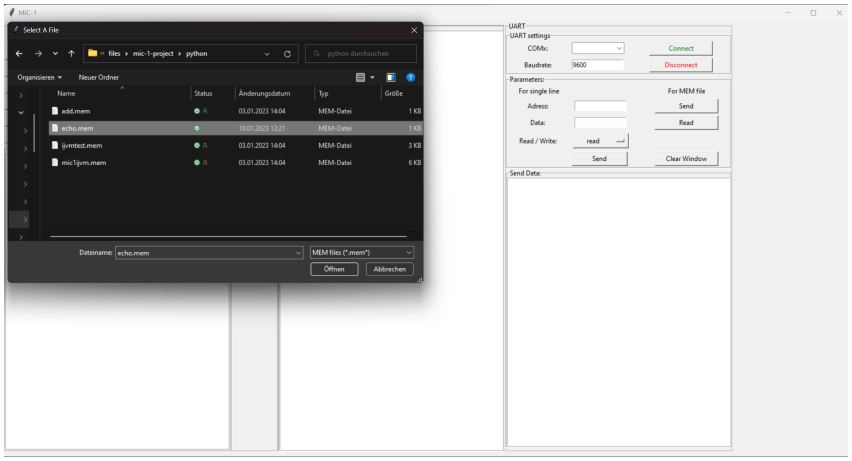

Abbildung 3.6: Search for Mem file in explorer

When the user presses the *Send* button for the MEM file the file data get send address by address (line by line) to the registers of the microcontroller.

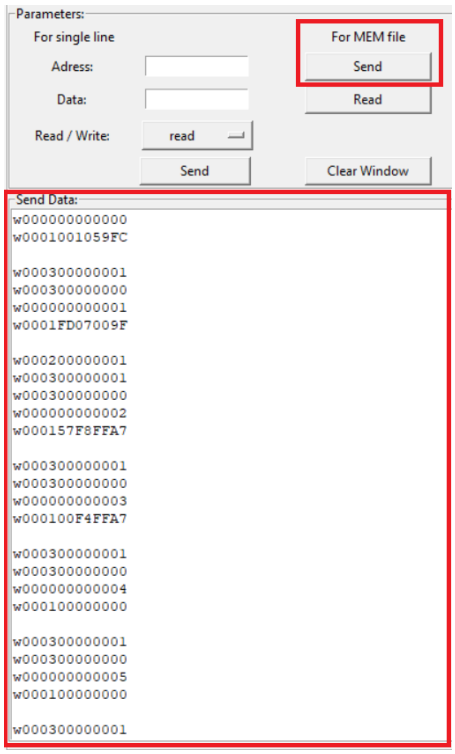

Abbildung 3.7: Send mem file to MIC-1

To read back the memory from the MIC-1 the user has to press the *Read* Button. The GUI reads back the data from every address and displays it in the *Table with content of register*.

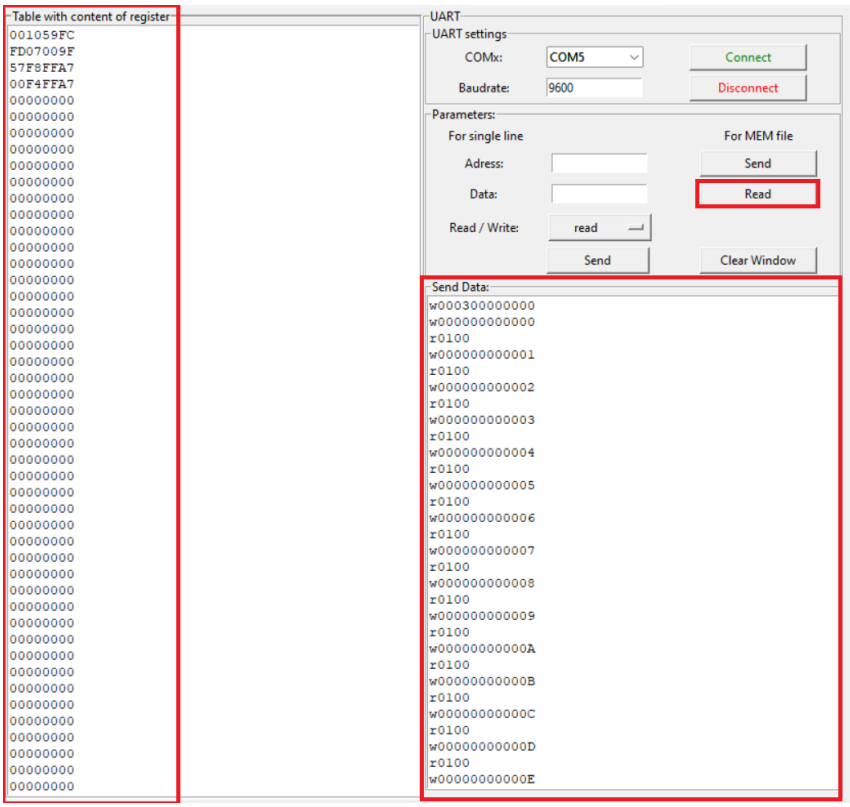

Abbildung 3.8: Read registers of MIC-1

3.0.3 Code

First the MIC1 class is defined with an __init__ method. In the __init__ method, several global variables (such as serial_data, filter_data, serial_object...) are defined. These variables are later used in different methods. Also a Tkinter window and its title is created in the __init__. The size of the window is set to fill the entire screen. In the last line the create_widgets method is called, which creates the buttons and fields that are displayed in the GUI. The create_widgets method is going to be described later.
The run method start the Tkinter main loop, which updates the GUI and waits for user input.

Listing 3.1: Initialisation and main loop

```

class MIC1:
    def __init__(self):
        #declare global variable
        self.serial_data = ''
        self.filter_data = ''
        self.update_period = 5
        self.serial_object = None
        self.file = None
        self.filename = None
        self.BRAMenable = "w000200000001"
        self.Wenable = "w000300000001"
        self.BRAMdisable = "w000200000000"
        self.Wdisable = "w000300000000"

        #create a window
        self.window = tk.Tk()
        self.window.title('MIC-1')
        w, h = self.window.winfo_screenwidth(), self.window.winfo_screenheight()
        self.window.geometry("%dx%d+0+0" % (w, h))
        self.uartState = False
        self.create_widgets()

    #mainloop
    def run(self):
        self.window.mainloop()
```

The connect method is used to establish a connection with the connected Baysy3 board over a serial port. It uses the Serial class from the 'serial' module to open a serial port specified by the user in the GUI and sets the baud rate. It also starts a new thread with the get_data method which runs in the background and continuously reads data from the device and updates the status window. After the device is connected, Connectedïs printed in the button status window, so the user knows the connection was established successfully.
To then close the serial port connection again a disconnect method is made. If the device is disconnected, "Disconnectedïs printed in the button status window.

Listing 3.2: UART Connection

```

    #connect uart
    def connect(self):
        global serial_object

        port = self.entryCOM.get()
        baud = self.entryBaudrate.get()
        self.serial_object = serial.Serial('COM' + str(port))
        self.serial_object.baudrate = baud

        t1 = threading.Thread(target=self.get_data)
```

```
        t1.daemon = True
        t1.start()

        connected = Label(self.buttontable, text="Connected")
        connected.grid()

    #disconnect uart
    def disconnect(self):
        global serial_object

        self.serial_object.close()

        disconnected = Label(self.buttontable, text="Disconnected")
        disconnected.grid()
```

The get_data method reads data from the device using the 'serial_object' variable defined earlier and inserts the data it in the status window in the GUI.

Listing 3.3: get_data method

```

    #get data from mic and display in status window
    def get_data(self):
        global serial_object
        global filter_data

        while (1):
            try:
                self.status.insert(END, self.serial_object.read(8))
                self.status.insert(END, "\n")


            except TypeError:
                pass
            self.window.update()
        self.status.insert(END, "\n")
        self.status.update()
```

The processButtonSend method is called when the user clicks the SSend"button. Before pressing the SSend"button the user also must select between read", "writeör "1 line".

If the readöption is selected, it sends the address and the read command to the device. The address is taken from the entry address widget and the read command is r0100". The commands are encoded as bytes and sent through the serial connection represented by the serial_object variable. The received data is logged in the status window. The sent commands are displayed in the text window.
If the "writeöption is selected, it sends both the address and the data to the device. The address is taken from the entry address widget and the data is taken from the entry data widget. First the address is sent and than the data. The sent commands are displayed in the text window.
If the "1 lineöption is selected, it sends the data that is typed in from the user as a single command to the device. The data is taken from the entry data widget. The sent command and any received data are logged in the text and status window.
In all three cases, the function uses the write method of the serial_object variable to send the commands and the read method to receive any data. The time.sleep function is used to wait for a short period of time to allow the device to process the command and respond.

Listing 3.4: processButtonSend

```
    #read from mic or write to mic, data and adress
    def processButtonSend(self):
        global serial_object
        global text
        selection = self.action.get()

        #if read is selected, send adress and read command
        if selection == "read":
            send_to_adress = self.entryadress.get()
            send_adress = send_to_adress
            self.serial_object.write(str.encode("w0000000000" + send_adress))
            self.text.insert(END, str.encode("w0000000000" + send_adress))
            self.text.insert(END, "\n")
            self.serial_object.write(str.encode("r0100"))
            self.text.insert(END, str.encode("r0100"))
            time.sleep(0.1)
            self.text.insert(END, "\n")
            self.status.insert(END, "\n")
            self.status.insert(END, self.serial_object.read())
            self.status.insert(END, "\n")

        #if write is selcted, send adress and data
        if selection == "write":
            send_data = self.entrydata.get()
            send_to_adress = self.entryadress.get()
            send_adress = send_to_adress
            self.serial_object.write(str.encode("w0000000000" + send_to_adress))
            self.text.insert(END, str.encode("w0000000000" + send_adress))
            self.text.insert(END, "\n")
            self.serial_object.write(str.encode("w0001" + send_data))
            self.text.insert(END, str.encode("w0001" + send_data))
            self.text.insert(END, "\n")

        #if 1 line is selected, just sends the one line that was entered in the
            data entry box
        if selection == "1 line":
            send_data = self.entrydata.get()
            self.serial_object.write(str.encode(send_data))
            self.text.insert(END, str.encode(send_data))
            self.text.insert(END, "\n")
            self.status.insert(END, "\n")
            self.status.insert(END, self.serial_object.read())
            self.status.insert(END, "\n")
```

The method called create_widgets is used to create and layout various elements (widgets). Buttons, labels, text boxes, entries and lableframes were used. Elements are created by calling a specific class constructor, such as tk.Label, tk.Entry, or tk.Button, and passing the root window as the first argument. Various properties of the widget can be set, such as its size, position, background color, font, and event handlers, by passing additional arguments to the constructor or by calling specific methods on the widget object. The grid method is used to position the element within the window, using row and column coordinates.

Listing 3.5: create$_w idgets$

```
    #creates all widgets for the gui
    def create_widgets(self):
        #create frame for buttons
        self.buttonframe = LabelFrame(self.window, labelanchor='nw', text="
            Buttons")
        self.buttonframe.grid(row=2, column=1, sticky=N + S)

        #global/write enable buttons
```

```
        self.globalenablebutton = Button(self.buttonframe, text='Global Enable',
                                fg='red', command=self.clickGlobalEnable)
        self.globalenablebutton.grid(row=3, column=1,sticky=W+E, padx=2, pady=2)

        self.globaldisablebutton = Button(self.buttonframe, text='Global Disable',
                                fg='blue', command=self.clickGlobalDisable)
        self.globaldisablebutton.grid(row=4, column=1,sticky=W+E, padx=2, pady=2)

        self.writeenablebutton = Button(self.buttonframe, text='Write Enable',
                                fg='violet', command=self.clickWriteEnable)
        self.writeenablebutton.grid(row=5, column=1,sticky=W+E, padx=2, pady=2)

        self.writedisablebutton = Button(self.buttonframe, text='Write Disable',
                                fg='purple', command=self.clickWriteDisable)
        self.writedisablebutton.grid(row=6, column=1,sticky=W+E, padx=2, pady=2)


        #call a python script
        self.callscript = Button(self.buttonframe, text="Open Script", command=self.fileDialog)
        self.callscript.grid(row=7, column=1, sticky=W+E, padx=2, pady=2)

        #Frame to display file code
        self.textframe = LabelFrame(self.buttonframe, labelanchor='nw',  text='File Code:')
        self.textframe.grid(row=8, column=1, sticky=S + SW)
        self.frameText = Text(self.textframe, width=50, height=30)
        self.frameText.grid(row=1, column=1, sticky=S + W)

        #lable to display path of the file that is open
        self.label = ttk.Label(self.buttonframe, text="")
        self.label.grid(row=0, column=1, sticky="ew", padx=5, pady=3)

        #create frame for the status of the button and to display number of lines from the imported file
        self.buttonstatusframe = LabelFrame(
            self.window, labelanchor='nw', text="Button Status", width=25)
        self.buttonstatusframe.grid(row=2, column=2, sticky=N + S + W + E)

        #canvas to write the status (disconnected/connected uart)
        self.buttontable = tk.Canvas(self.buttonstatusframe,
                                     width=100, height=400)
        self.buttontable.grid(column=2, row=1, padx=5, pady=5)

        #space to display number of lines from the imported file
        self.numberlines = Label(self.buttontable, text='')
        self.numberlines.grid()

        #create frame to display the content of the register
        self.statusframe = LabelFrame(self.window, labelanchor='nw',
                                     text="Table with content of register", width=50, bd=3)
        self.statusframe.grid(row=2, column=3, sticky=N + S)

        #textbox to display status of register
        self.status = Text(self.statusframe, width=50, height=40)
        self.status.grid(row=1, column=1, sticky=N + S)

        # create frame for uart
        self.dataframe = LabelFrame(self.window, text='UART', width=50)
        self.dataframe.grid(column=5, row=2, sticky=N+W+S)

        #frame for uart settings (com/baudrate)
        self.uartframe = LabelFrame(self.dataframe, text='UART settings',width=50)
```

```
        self.uartframe.grid(column=1, row=1, sticky=N+W+S+E)

        #lable and entry for the COM
        self.labelCOM = Label(self.uartframe, text="COMx: ", width=15)
        COM = StringVar()
        self.entryCOM = Entry(self.uartframe, textvariable=COM, width=15)
        self.labelCOM.grid(row=1, column=1, padx=2, pady=2)
        self.entryCOM.grid(row=1, column=2, padx=2, pady=2)

        #lable and entry for the baudrate
        self.labelBaudrate = Label(self.uartframe, text="Baudrate: ", width=15)
        self.Baudrate = StringVar(value="9600")
        self.entryBaudrate = Entry(self.uartframe, textvariable=self.Baudrate,
            width=15)
        self.labelBaudrate.grid(row=2, column=1, padx=2, pady=2)
        self.entryBaudrate.grid(row=2, column=2, padx=2, pady=2)

        #blank lable between buttons and entry boxes
        self.blank_label = Label(self.uartframe, text="", width=5)
        self.blank_label.grid(row=1, column=3, columnspan=1)

        #connect button for uart
        self.buttonC = tk.Button(self.uartframe, text="Connect", command=self.
            connect, width=15)
        self.buttonC.grid(row=1, column=4, padx=2, pady=2, sticky=E)

        #connect button for uart
        self.buttonD = tk.Button(self.uartframe, text="Disconnect", command=self
            .disconnect, width=15)
        self.buttonD.grid(row=2, column=4, padx=2, pady=2, sticky=E)

        #frame to communicate (write/read) with the MIC
        self.adressframe = LabelFrame(self.dataframe, text='Parameters:', width
            =50)
        self.adressframe.grid(column=1, row=3, sticky=N+W+S+E)

        #lable read and write
        self.labelaction = Label(self.adressframe, text="Read / Write: ", width
            =15)
        self.labelaction.grid(row=4, column=1, padx=2, pady=2)

        #to choose if read from or write to the MIC or jus send one line
        self.choices = ['read', 'write', '1 line']
        self.action = StringVar()
        self.action.set('read')
        self.actiondata = OptionMenu(self.adressframe, self.action, *self.
            choices)
        self.actiondata.grid(row=4, column=2, padx=2, pady=2)

        #button to send file
        self.buttonSendmem = Button(self.adressframe, text="Send mem", command=
            self.processButtonSendmem, width=15)
        self.buttonSendmem.grid(row=5, column=4, padx=2, pady=2, sticky=tk.W)

        #read button
        self.buttonRead = Button(self.adressframe, text="Read", command=self.
            processButtonRead, width=15)
        self.buttonRead.grid(row=6, column=4, padx=2, pady=2, sticky=tk.W)

        #button to send adress/data
        self.buttonSend2 = Button(self.adressframe, text="Send", command=self.
            processButtonSend, width=15)
        self.buttonSend2.grid(row=4, column=4, padx=2, pady=2, sticky=tk.W)

        #blank lable to keeo the gui clean
        self.blank_label2 = Label(self.adressframe, text="", width=5)
        self.blank_label2.grid(row=1, column=3, columnspan=1)
```

```

        #lable and entry box for adress
        self.labeladress = Label(self.adressframe, text="Adress: ", width=15)
        self.adress = StringVar()
        self.entryadress = Entry(self.adressframe, textvariable=self.adress,
            width=15)
        self.labeladress.grid(row=5, column=1, padx=2, pady=2)
        self.entryadress.grid(row=5, column=2, padx=2, pady=2)

        #lable and entry box for data
        self.labelData = Label(self.adressframe, text="Data: ", width=15)
        self.Data = StringVar()
        self.entrydata = Entry(self.adressframe, textvariable=self.Data, width
            =15)
        self.labelData.grid(row=6, column=1, padx=2, pady=2)
        self.entrydata.grid(row=6, column=2, padx=2, pady=2)

        #button to clear the status and the send data window
        self.buttonRead = Button(self.adressframe, text="Clear Window", command=
            self.processButtonDelete, width=15)
        self.buttonRead.grid(row=7, column=4, padx=2, pady=2, sticky=tk.W)

        #frame to display send data
        self.labelOutText = LabelFrame(self.dataframe, text="Send Data:", width
            =50)
        self.labelOutText.grid(row=6, column=1, sticky=S+W)
        self.text = Text(self.labelOutText, width=50, height=26)
        self.text.grid(row=7, column=1,sticky=S+W+E)
```

3.0.4 Send data

The function called processButtonSendmem sends the contents of a file to the serial object.
The function operates as a loop that runs num_lines times. num_lines Is the variable that holds the number of lines that the file has that is going to be send. For each iteration, the following steps are performed:
The address of the memory location to be written to is set by concatenating "0"with the current iteration number, padded with zeros to a total of 2 characters. First the write command (the string "w0000000000followed by the current address) is sent. The file is opened and the line of the file corresponding to the current iteration is read and stored in the memdata variable. The contents of memdata are converted to uppercase and sent to the serial object as a write command (the string "w0001followed by the uppercase contents of memdata). Because the MIC needs uppercase letters. BRAMenable is send to enable the BRAM. Then the Wenable and Wdisable commands are sent. The address is incremented and converted to uppercase hexadecimal format. The resulting string is padded with zeros to a total of 2 characters. Then the file is closed. After the loop has completed, the address is set to 0 and the file is closed.

Listing 3.6: processButtonSendmem

```

    #send the whole (mem) file
    def processButtonSendmem(self):
        global file
        global filename
        global serial_object
        global BRAMenable
        global BRAMdisable
        global Wenable
        global Wdisable
        global num_lines

```

```
        adress = "0".zfill(2)
        for intadress in range(1,num_lines):
            self.serial_object.write(str.encode("w0000000000" + adress))
            self.text.insert(END, str.encode("w0000000000" + adress))
            self.text.insert(END, "\n")
            file = open(filename, 'r')
            memdata = file.readlines()[intadress]
            memdataupper = str.upper(memdata)
            self.serial_object.write(str.encode("w0001" + (memdataupper)))
            self.text.insert(END, str.encode("w0001" + (memdataupper)))
            self.text.insert(END, "\n")
            if adress == "01":
                self.serial_object.write(str.encode(self.BRAMenable))
                self.text.insert(END, str.encode(self.BRAMenable))
                self.text.insert(END, "\n")
            self.serial_object.write(str.encode(self.Wenable))
            self.text.insert(END, str.encode(self.Wenable))
            self.text.insert(END, "\n")
            self.serial_object.write(str.encode(self.Wdisable))
            self.text.insert(END, str.encode((self.Wdisable)))
            self.text.insert(END, "\n")
            intconadress = int(adress, base=16)
            intconadress +=1
            if intconadress % 16 == 0:
                hexa = hex(intconadress)
                hexadress = hexa.strip('0x')
                hadress = str(hexadress + "0")
            else:
                hexa = hex(intconadress)
                hexadress = hexa.strip('0x')
                hadress = str(hexadress)
            uppercaseadress = str.upper(hadress)
            adress = uppercaseadress.zfill(2)
            file.close()
        adress = "0"
        file.close()
```

3.0.5 Read data

The function processButtonRead reads the registers of the MIC-1. For that the function sends the Wdisable command in the beginning.
The function operates as a loop that runs num_lines times. num_lines Is the variable that holds the number of registers the function has to read out. It is given by the function processButtonSendmem.For each iteration, the following steps are performed:
The address of the memory location to be written to is set by concatenating "0"with the current iteration number, padded with zeros to a total of 2 characters. First the write command (the string "w0000000000followed by the current address) is sent. To read out the registers the read command (the string r0100") is sent. The address is incremented and converted to uppercase hexadecimal format. The resulting string is padded with zeros to a total of 2 characters. After the loop has completed, the address is set to 0.

Listing 3.7: processButtonRead

```

    global BRAMenable
        global Wenable
        global serial_object
        global num_lines

        self.serial_object.write(str.encode(self.Wdisable))
```

```
        self.text.insert(END, str.encode((self.Wdisable)))
        self.text.insert(END, "\n")

        adress = "0".zfill(2)
        for i in range(1,num_lines):
            self.serial_object.write(str.encode("w0000000000" + adress))
            self.text.insert(END, str.encode("w0000000000" + adress))
            self.text.insert(END, "\n")
            self.serial_object.write(str.encode("r0100"))
            self.text.insert(END, str.encode("r0100"))
            self.text.insert(END, "\n")
            #self.status.insert(END, "\n")
            #self.status.insert(END, self.serial_object.readline())
            time.sleep(0.1)

            intconadress = int(adress, base=16)
            intconadress +=1
            if intconadress % 16 == 0:
                hexa = hex(intconadress)
                hexadress = hexa.strip('0x')
                hadress = str(hexadress + "0")
            else:
                hexa = hex(intconadress)
                hexadress = hexa.strip('0x')
                hadress = str(hexadress)
            uppercaseadress = str.upper(hadress)
            adress = uppercaseadress.zfill(2)
        adress = "0"
```

A VGA code

A.1 Original VGA Controller code

Listing A.1: Input, outputs and all internal variables and signals

```
module num_capture_4bit
    //Input, Output definition and the internal signals
    (
    input wire iClk, iRst, iIncr, iStop, iExtra, iSw0, iSw1,

    output wire[9:0] oAddr,
    output wire[11:0] oData,
    output wire oWe);

    reg [11:0]  rNum_Curr, rNum_Next;
    reg rWeN, rWe, rToggle_Curr, rToggle_Next;
    reg [9:0]   rAddr_Next, rAddr_Curr;
    reg[4:0] rFSM_Curr, wFSM_Next;

    //State Definition
    localparam sInit    = 4'b0000;
    localparam sIdle    = 4'b0001;
    localparam sPush    = 4'b0010;
    localparam sIncA    = 4'b0011;
    localparam sIncB    = 4'b0100;
    localparam sStop    = 4'b0101;
    localparam sStopInc = 4'b0110;
    localparam sTog     = 4'b0111;
    localparam sTogInc  = 4'b1000;
    localparam sRst     = 4'b1001;
    localparam sSpace   = 4'b1010;
    localparam sSpaceInc= 4'b1011;
```

```
    localparam sBreak    = 4'b1100;
    localparam sBreakInc= 4'b1101;
    localparam sDelete   = 4'b1110;
    localparam sDelInc   = 4'b1111;
```

Listing A.2: Next State Logic

```
    //State Register without reset
    always @(posedge iClk)
    begin
        rFSM_Curr <= wFSM_Next;
    end

    //Next State Logic
    always @(posedge iClk)
    begin
        case (rFSM_Curr)

            sInit    :   if (rAddr_Curr > 0)
                             wFSM_Next <= sInit;
                         else
                             wFSM_Next <= sIdle;

            sIdle    :   if (iStop == 0 && iIncr == 0 && iRst == 0 && iExtra ==
                 0)
                             wFSM_Next <= sIdle;
                         else if (iRst == 1)
                             wFSM_Next <= sRst;
                         else if (iIncr == 1)
                             wFSM_Next <= sPush;
                         else if (iStop == 1)
                             wFSM_Next <= sStop;
                         else if (iExtra == 1 && iSw0 == 0 && iSw1 == 0)
                             wFSM_Next <= sTog;
                         else if (iExtra == 1 && iSw0 == 1 && iSw1 == 1)
                             wFSM_Next <= sSpace;
                         else if (iExtra == 1 && iSw0 == 0 && iSw1 == 1)
                             wFSM_Next <= sBreak;
                         else if (iExtra == 1 && iSw0 == 1 && iSw1 == 0)
                             wFSM_Next <= sDelete;

            sPush    :   if (iIncr == 1)
                             wFSM_Next <= sPush;
                         else if (iIncr == 0)
                             if (rToggle_Curr == 0)
                                 wFSM_Next <= sIncA;
                             else
                                 wFSM_Next <= sIncB;

            sIncA    :   wFSM_Next <= sIdle;

            sIncB    :   wFSM_Next <= sIdle;

            sStop    :   if (iStop == 1)
                             wFSM_Next <= sStop;
                         else if (iStop == 0)
                             wFSM_Next <= sStopInc;

            sStopInc :   wFSM_Next <= sIdle;

            sTog     :   if (iExtra == 1 && iSw0 == 0 && iSw1 == 0)
                             wFSM_Next <= sTog;
                         else if (iExtra == 0 && iSw0 == 0 && iSw1 == 0)
                             wFSM_Next <= sTogInc;

```

```
            sTogInc  :   wFSM_Next <= sIdle;

            sRst     :   wFSM_Next <= sInit;

            sSpace   :   if( iExtra == 1 && iSw0 == 1 && iSw1 == 1)
                             wFSM_Next <= sSpace;
                         else
                             wFSM_Next <= sSpaceInc;

            sSpaceInc : wFSM_Next <= sIdle;

            sBreak   :    if (iExtra == 1 && iSw0 == 0 && iSw1 == 1)
                              wFSM_Next <= sBreak;
                          else
                              wFSM_Next <= sBreakInc;

            sBreakInc : wFSM_Next <= sIdle;

            sDelete : if (iExtra == 1 && iSw0 == 1 && iSw1 == 0)
                        wFSM_Next <= sDelete;
                      else
                        wFSM_Next <= sDelInc;

            sDelInc : wFSM_Next <= sIdle;

            default  :   wFSM_Next <= sIdle;

        endcase
    end
```

Listing A.3: Output Logic

```
    //Output Logic
    //Only in function of states
   always @(posedge iClk)
    begin
    case (rFSM_Curr)
        sIncA    :   rAddr_Next <= rAddr_Curr;

        sIncB    :   rAddr_Next <= rAddr_Curr;

        sInit    :   if ( rAddr_Curr > 0 )
                         rAddr_Next <= rAddr_Curr - 1;
                     else
                         rAddr_Next <= 0;

        sIdle    :   rAddr_Next <= rAddr_Curr;

        sPush    :   rAddr_Next <= rAddr_Curr;

        sStop    :   rAddr_Next <= rAddr_Curr;

        sStopInc:    if (rAddr_Curr < 599)
                         rAddr_Next <= rAddr_Curr + 1;
                     else
                         rAddr_Next <= 0;

        sRst     :   rAddr_Next <= 600;

        sSpace   :   rAddr_Next <= rAddr_Curr;

        sSpaceInc:   if (rAddr_Curr < 598)
                         rAddr_Next <= rAddr_Curr + 2;
                     else
                         rAddr_Next <= 0;
```

```

        sBreak   :   rAddr_Next <= rAddr_Curr;

        sBreakInc : if (rAddr_Curr >= 560)
                        rAddr_Next <= 0;
                    else
                        rAddr_Next <= rAddr_Curr + 40 - (rAddr_Curr % 40);

        sDelete : rAddr_Next <= rAddr_Curr;

        sDelInc : if (rAddr_Curr == 0)
                    rAddr_Next <= rAddr_Curr;
                   else
                    rAddr_Next <= rAddr_Curr - 1;

        default :   rAddr_Next <= rAddr_Curr;
    endcase
    end
```

Listing A.4: State Register

```
always @(posedge iClk)
    begin
    case (rFSM_Curr)
        sIncA     :  if ( rNum_Curr == 0 )
                        rNum_Next <= 512;
                    else if ( rNum_Curr >= 512 && rNum_Curr <= 768 )
                        rNum_Next <= rNum_Curr + 32;
                    else if ( rNum_Curr == 800 )
                        rNum_Next <= 1056;
                    else if ( rNum_Curr < 1216 )
                        rNum_Next <= rNum_Curr + 32;

                  // in case of switch while a lowercase letter is on screen
                    else if ( rNum_Curr >= 2080 && rNum_Curr < 2240)
                        rNum_Next <= rNum_Curr - 2080 + 1056;
                    else
                        rNum_Next <= 512;

      // Alternate way of displaying  numbers > 9, enabled by iSwitchA)
         sIncB   :  if ( rNum_Curr == 0 )
                        rNum_Next <= 512;
                    else if ( rNum_Curr >= 512 && rNum_Curr <= 768 )
                        rNum_Next <= rNum_Curr + 32;
                    else if ( rNum_Curr == 800 )
                        rNum_Next <= 2080;

                // in case of switch while a UPPERcase letter is on screen
                    else if ( rNum_Curr >= 1056 && rNum_Curr < 1216)
                        rNum_Next <= rNum_Curr + 2080 - 1056;
                    else if ( rNum_Curr <= 2208 )
                        rNum_Next <= rNum_Curr + 32;
                    else
                        rNum_Next <= 512;

        sInit    :   rNum_Next <= 0;

        sIdle    :   rNum_Next <= rNum_Curr;

        sPush    :   rNum_Next <= rNum_Curr;

        sStop    :   rNum_Next <= rNum_Curr;

        sStopInc:    rNum_Next <= 512;

```

```
        sRst     :   rNum_Next <= 0;

        sSpace   :   rNum_Next <= rNum_Curr;

        sSpaceInc:  rNum_Next <= 0;

        sBreak   :   rNum_Next <= rNum_Curr;

        sBreakInc : rNum_Next <= 0;

        sDelete: rNum_Next <= 0;

        sDelInc: rNum_Next <= 0;

        default  :   rNum_Next <= rNum_Curr;

    endcase
    end

    always @(posedge iClk)
    begin
        case(rFSM_Curr)

        sTogInc :   if (rToggle_Curr == 0)
                        rToggle_Next <= 1;
                    else
                        rToggle_Next <= 0;

        default  :   rToggle_Next <= rToggle_Next;

        endcase
    end

    always @(posedge iClk)
    begin
        rAddr_Curr   <= rAddr_Next;
        rNum_Curr    <= rNum_Next;
        rToggle_Curr <= rToggle_Next;
    end

    assign oAddr = rAddr_Curr;
    assign oWe   = (rFSM_Curr == sIdle || rFSM_Curr == sInit || rFSM_Curr ==
        sDelInc) ? 1 : 0;
    assign oData = rNum_Curr;

endmodule
```

A.2 New VGA Controller code

Listing A.5: New input, outputs and all internal variables and signals

```
module num_capture_4bit

    (
    input wire iClk, iRst, iSwUpdate, //change to switch
    input wire[47:0] iCPP, iH,
                     iLV, iMAR,
                     iMDR, iOPC,
                     iPC, iSP,
                     iTOS, iData,
    input wire[11:0] iMBR,

    output wire[9:0]  oAddr,
```

```
    output wire[11:0] oData,
    output wire oWe
    );
    reg [11:0]  rNum_Curr, rNum_Next;
    reg         rWeN, rWe;
    reg [9:0]   rAddr_Next, rAddr_Curr;
    reg[4:0]    rFSM_Curr, wFSM_Next;

    //State Definition
    localparam sInit     = 4'b0000;
    localparam sIdle     = 4'b0001;
    localparam sUpdate0  = 4'b0010;
    localparam sUpdate1  = 4'b0011;
    localparam sUpdate2  = 4'b0100;
    localparam sUpdate3  = 4'b0101;
    localparam sUpdate4  = 4'b0110;
    localparam sUpdate5  = 4'b0111;
    localparam sUpdate6  = 4'b1000;
    localparam sUpdate7  = 4'b1001;
    localparam sUpdate8  = 4'b1010;
    localparam sUpdate9  = 4'b1011;
    localparam sUpdate10 = 4'b1100;
    localparam sRst      = 4'b1101;
```

Listing A.6: New Next State Logic

```
    //State Register without reset
    always @(posedge iClk)
    begin
        rFSM_Curr <= wFSM_Next;
    end

    //Next State Logic
    always @(posedge iClk)
    begin
        case (rFSM_Curr)

            sRst     :   wFSM_Next <= sInit;

            sInit    :   if (rAddr_Curr < 497)
                            wFSM_Next <= sInit;
                        else
                            wFSM_Next <= sIdle;

            sIdle    :   if (iSwUpdate == 1)
                            wFSM_Next <= sUpdate0;
                        else if (iRst == 1)
                            wFSM_Next <= sRst;
                        else
                            wFSM_Next <= sIdle;

            sUpdate0 :   if (rAddr_Curr == 97) //End of HEX line
                            wFSM_Next <= sUpdate1;
                        else
                            wFSM_Next <= sUpdate0;

            sUpdate1 :   if (rAddr_Curr == 137) //End of HEX line
                            wFSM_Next <= sUpdate2;
                        else
                            wFSM_Next <= sUpdate1;

            sUpdate2 :   if (rAddr_Curr == 177) //End of HEX line
                            wFSM_Next <= sUpdate3;
                        else
                            wFSM_Next <= sUpdate2;
```

```
            sUpdate3 :    if (rAddr_Curr == 217) //End of HEX line
                             wFSM_Next <= sUpdate4;
                          else
                             wFSM_Next <= sUpdate3;

            sUpdate4 :    if (rAddr_Curr == 257) //End of HEX line
                             wFSM_Next <= sUpdate5;
                          else
                             wFSM_Next <= sUpdate4;

            sUpdate5 :    if (rAddr_Curr == 297) //End of HEX line
                             wFSM_Next <= sUpdate6;
                          else
                             wFSM_Next <= sUpdate5;

            sUpdate6 :    if (rAddr_Curr == 337) //End of HEX line
                             wFSM_Next <= sUpdate7;
                          else
                             wFSM_Next <= sUpdate6;

            sUpdate7 :    if (rAddr_Curr == 377) //End of HEX line
                             wFSM_Next <= sUpdate8;
                          else
                             wFSM_Next <= sUpdate7;

            sUpdate8 :    if (rAddr_Curr == 417) //End of HEX line
                             wFSM_Next <= sUpdate9;
                          else
                             wFSM_Next <= sUpdate8;

            sUpdate9 :    if (rAddr_Curr == 457) //End of HEX line
                             wFSM_Next <= sUpdate10;
                          else
                             wFSM_Next <= sUpdate9;

            sUpdate10 :    if (rAddr_Curr == 497) //End of HEX line
                             wFSM_Next <= sIdle;
                          else
                             wFSM_Next <= sUpdate10;

            default   :   wFSM_Next <= sIdle;

        endcase
    end
```

Listing A.7: New Output Logic

```
    //Output Logic
    //only in function of states
    always @(posedge iClk)
    begin
        case (rFSM_Curr)

            sRst      :   rAddr_Next <= 0;

            //Only delete the data not the static values
            sInit     :   if (rAddr_Curr < 91)
                             rAddr_Next <= 91;
                          else if (rAddr_Curr == 98)
                             rAddr_Next <= 131;
                          else if (rAddr_Curr == 138)
                             rAddr_Next <= 171;
                          else if (rAddr_Curr == 178)
                             rAddr_Next <= 211;
```

```
                        else if (rAddr_Curr == 218)
                           rAddr_Next <= 251;
                        else if (rAddr_Curr == 258)
                           rAddr_Next <= 291;
                        else if (rAddr_Curr == 298)
                           rAddr_Next <= 331;
                        else if (rAddr_Curr == 338)
                           rAddr_Next <= 371;
                        else if (rAddr_Curr == 378)
                           rAddr_Next <= 411;
                        else if (rAddr_Curr == 418)
                           rAddr_Next <= 451;
                        else if (rAddr_Curr == 458)
                           rAddr_Next <= 491;
                        else
                           rAddr_Next = rAddr_Curr + 1;

            sIdle     :   rAddr_Next <= 0;

            sUpdate0  :   if (rAddr_Curr < 91)
                           rAddr_Next <= 91;
                        else
                           rAddr_Next <= rAddr_Curr + 1;

            sUpdate1  :   if (rAddr_Curr < 131)
                           rAddr_Next <= 131;
                        else
                           rAddr_Next <= rAddr_Curr + 1;

            sUpdate2  :   if (rAddr_Curr < 171)
                           rAddr_Next <= 171;
                        else
                           rAddr_Next <= rAddr_Curr + 1;

            sUpdate3  :   if (rAddr_Curr < 211)
                           rAddr_Next <= 211;
                        else
                           rAddr_Next <= rAddr_Curr + 1;

            sUpdate4  :   if (rAddr_Curr < 251)
                           rAddr_Next <= 251;
                        else
                           rAddr_Next <= rAddr_Curr + 1;

            sUpdate5  :   if (rAddr_Curr < 291)
                           rAddr_Next <= 291;
                        else
                           rAddr_Next <= rAddr_Curr + 1;

            sUpdate6  :   if (rAddr_Curr < 331)
                           rAddr_Next <= 331;
                        else
                           rAddr_Next <= rAddr_Curr + 1;

            sUpdate7  :   if (rAddr_Curr < 371)
                           rAddr_Next <= 371;
                        else
                           rAddr_Next <= rAddr_Curr + 1;

            sUpdate8  :   if (rAddr_Curr < 411)
                           rAddr_Next <= 411;
                        else
                           rAddr_Next <= rAddr_Curr + 1;

            sUpdate9  :   if (rAddr_Curr < 451)
                           rAddr_Next <= 451;
                        else
```

```
                                rAddr_Next <= rAddr_Curr + 1;

            sUpdate10 :    if (rAddr_Curr < 491)
                                rAddr_Next <= 491;
                            else
                                rAddr_Next <= rAddr_Curr + 1;

            default   :    rAddr_Next <= rAddr_Curr;

        endcase
    end
```

Listing A.8: New State Register

```
always @(posedge iClk)
    begin
        case (rFSM_Curr)

            sRst      :    rNum_Next <= 448;

            sInit     :    rNum_Next <= 448;

            sIdle     :    rNum_Next <= 0;

            sUpdate0  :    if (rAddr_Curr < 91)
                                rNum_Next <= {1'b0, iCPP[47:42], 5'b00000};
                            else if (rAddr_Curr == 91)
                                rNum_Next <= {1'b0, iCPP[41:36], 5'b00000};
                            else if (rAddr_Curr == 92)
                                rNum_Next <= {1'b0, iCPP[35:30], 5'b00000};
                            else if (rAddr_Curr == 93)
                                rNum_Next <= {1'b0, iCPP[29:24], 5'b00000};
                            else if (rAddr_Curr == 94)
                                rNum_Next <= {1'b0, iCPP[23:18], 5'b00000};
                            else if (rAddr_Curr == 95)
                                rNum_Next <= {1'b0, iCPP[17:12], 5'b00000};
                            else if (rAddr_Curr == 96)
                                rNum_Next <= {1'b0, iCPP[11:6], 5'b00000};
                            else if (rAddr_Curr == 97)
                                rNum_Next <= {1'b0, iCPP[5:0], 5'b00000};

            sUpdate1  :    if (rAddr_Curr < 131)
                                rNum_Next <= {1'b0, iH[47:42], 5'b00000};
                            else if (rAddr_Curr == 131)
                                rNum_Next <= {1'b0, iH[41:36], 5'b00000};
                            else if (rAddr_Curr == 132)
                                rNum_Next <= {1'b0, iH[35:30], 5'b00000};
                            else if (rAddr_Curr == 133)
                                rNum_Next <= {1'b0, iH[29:24], 5'b00000};
                            else if (rAddr_Curr == 134)
                                rNum_Next <= {1'b0, iH[23:18], 5'b00000};
                            else if (rAddr_Curr == 135)
                                rNum_Next <= {1'b0, iH[17:12], 5'b00000};
                            else if (rAddr_Curr == 136)
                                rNum_Next <= {1'b0, iH[11:6], 5'b00000};
                            else if (rAddr_Curr == 137)
                                rNum_Next <= {1'b0, iH[5:0], 5'b00000};

            sUpdate2  :    if (rAddr_Curr < 171)
                                rNum_Next <= {1'b0, iLV[47:42], 5'b00000};
                            else if (rAddr_Curr == 171)
                                rNum_Next <= {1'b0, iLV[41:36], 5'b00000};
                            else if (rAddr_Curr == 172)
                                rNum_Next <= {1'b0, iLV[35:30], 5'b00000};
                            else if (rAddr_Curr == 173)
```

```
                        rNum_Next <= {1'b0, iLV[29:24], 5'b00000};
                     else if (rAddr_Curr == 174)
                        rNum_Next <= {1'b0, iLV[23:18], 5'b00000};
                     else if (rAddr_Curr == 175)
                        rNum_Next <= {1'b0, iLV[17:12], 5'b00000};
                     else if (rAddr_Curr == 176)
                        rNum_Next <= {1'b0, iLV[11:6], 5'b00000};
                     else if (rAddr_Curr == 177)
                        rNum_Next <= {1'b0, iLV[5:0], 5'b00000};

         sUpdate3 :  if (rAddr_Curr < 211)
                        rNum_Next <= {1'b0, iMAR[47:42], 5'b00000};
                     else if (rAddr_Curr == 211)
                        rNum_Next <= {1'b0, iMAR[41:36], 5'b00000};
                     else if (rAddr_Curr == 212)
                        rNum_Next <= {1'b0, iMAR[35:30], 5'b00000};
                     else if (rAddr_Curr == 213)
                        rNum_Next <= {1'b0, iMAR[29:24], 5'b00000};
                     else if (rAddr_Curr == 214)
                        rNum_Next <= {1'b0, iMAR[23:18], 5'b00000};
                     else if (rAddr_Curr == 215)
                        rNum_Next <= {1'b0, iMAR[17:12], 5'b00000};
                     else if (rAddr_Curr == 216)
                        rNum_Next <= {1'b0, iMAR[11:6], 5'b00000};
                     else if (rAddr_Curr == 217)
                        rNum_Next <= {1'b0, iMAR[5:0], 5'b00000};

         sUpdate4 :  if (rAddr_Curr == 256)
                        rNum_Next <= {1'b0, iMBR[11:6], 5'b00000};
                     else if (rAddr_Curr == 257)
                        rNum_Next <= {1'b0, iMBR[5:0], 5'b00000};
                     else
                        rNum_Next <= 0;

         sUpdate5 :  if (rAddr_Curr < 291)
                        rNum_Next <= {1'b0, iMDR[47:42], 5'b00000};
                     else if (rAddr_Curr == 291)
                        rNum_Next <= {1'b0, iMDR[41:36], 5'b00000};
                     else if (rAddr_Curr == 292)
                        rNum_Next <= {1'b0, iMDR[35:30], 5'b00000};
                     else if (rAddr_Curr == 293)
                        rNum_Next <= {1'b0, iMDR[29:24], 5'b00000};
                     else if (rAddr_Curr == 294)
                        rNum_Next <= {1'b0, iMDR[23:18], 5'b00000};
                     else if (rAddr_Curr == 295)
                        rNum_Next <= {1'b0, iMDR[17:12], 5'b00000};
                     else if (rAddr_Curr == 296)
                        rNum_Next <= {1'b0, iMDR[11:6], 5'b00000};
                     else if (rAddr_Curr == 297)
                        rNum_Next <= {1'b0, iMDR[5:0], 5'b00000};

         sUpdate6 :  if (rAddr_Curr < 331)
                        rNum_Next <= {1'b0, iOPC[47:42], 5'b00000};
                     else if (rAddr_Curr == 331)
                        rNum_Next <= {1'b0, iOPC[41:36], 5'b00000};
                     else if (rAddr_Curr == 332)
                        rNum_Next <= {1'b0, iOPC[35:30], 5'b00000};
                     else if (rAddr_Curr == 333)
                        rNum_Next <= {1'b0, iOPC[29:24], 5'b00000};
                     else if (rAddr_Curr == 334)
                        rNum_Next <= {1'b0, iOPC[23:18], 5'b00000};
                     else if (rAddr_Curr == 335)
                        rNum_Next <= {1'b0, iOPC[17:12], 5'b00000};
                     else if (rAddr_Curr == 336)
                        rNum_Next <= {1'b0, iOPC[11:6], 5'b00000};
                     else if (rAddr_Curr == 337)
                        rNum_Next <= {1'b0, iOPC[5:0], 5'b00000};
```

```
119
120             sUpdate7 :    if (rAddr_Curr < 371)
121                              rNum_Next <= {1'b0, iPC[47:42], 5'b00000};
122                           else if (rAddr_Curr == 371)
123                              rNum_Next <= {1'b0, iPC[41:36], 5'b00000};
124                           else if (rAddr_Curr == 372)
125                              rNum_Next <= {1'b0, iPC[35:30], 5'b00000};
126                           else if (rAddr_Curr == 373)
127                              rNum_Next <= {1'b0, iPC[29:24], 5'b00000};
128                           else if (rAddr_Curr == 374)
129                              rNum_Next <= {1'b0, iPC[23:18], 5'b00000};
130                           else if (rAddr_Curr == 375)
131                              rNum_Next <= {1'b0, iPC[17:12], 5'b00000};
132                           else if (rAddr_Curr == 376)
133                              rNum_Next <= {1'b0, iPC[11:6], 5'b00000};
134                           else if (rAddr_Curr == 377)
135                              rNum_Next <= {1'b0, iPC[5:0], 5'b00000};
136
137             sUpdate8 :    if (rAddr_Curr < 411)
138                              rNum_Next <= {1'b0, iSP[47:42], 5'b00000};
139                           else if (rAddr_Curr == 411)
140                              rNum_Next <= {1'b0, iSP[41:36], 5'b00000};
141                           else if (rAddr_Curr == 412)
142                              rNum_Next <= {1'b0, iSP[35:30], 5'b00000};
143                           else if (rAddr_Curr == 413)
144                              rNum_Next <= {1'b0, iSP[29:24], 5'b00000};
145                           else if (rAddr_Curr == 414)
146                              rNum_Next <= {1'b0, iSP[23:18], 5'b00000};
147                           else if (rAddr_Curr == 415)
148                              rNum_Next <= {1'b0, iSP[17:12], 5'b00000};
149                           else if (rAddr_Curr == 416)
150                              rNum_Next <= {1'b0, iSP[11:6], 5'b00000};
151                           else if (rAddr_Curr == 417)
152                              rNum_Next <= {1'b0, iSP[5:0], 5'b00000};
153
154             sUpdate9 :    if (rAddr_Curr < 451)
155                              rNum_Next <= {1'b0, iTOS[47:42], 5'b00000};
156                           else if (rAddr_Curr == 451)
157                              rNum_Next <= {1'b0, iTOS[41:36], 5'b00000};
158                           else if (rAddr_Curr == 452)
159                              rNum_Next <= {1'b0, iTOS[35:30], 5'b00000};
160                           else if (rAddr_Curr == 453)
161                              rNum_Next <= {1'b0, iTOS[29:24], 5'b00000};
162                           else if (rAddr_Curr == 454)
163                              rNum_Next <= {1'b0, iTOS[23:18], 5'b00000};
164                           else if (rAddr_Curr == 455)
165                              rNum_Next <= {1'b0, iTOS[17:12], 5'b00000};
166                           else if (rAddr_Curr == 456)
167                              rNum_Next <= {1'b0, iTOS[11:6], 5'b00000};
168                           else if (rAddr_Curr == 457)
169                              rNum_Next <= {1'b0, iTOS[5:0], 5'b00000};
170
171             sUpdate10 :    if (rAddr_Curr < 491)
172                              rNum_Next <= {1'b0, iData[47:42], 5'b00000};
173                           else if (rAddr_Curr == 491)
174                              rNum_Next <= {1'b0, iData[41:36], 5'b00000};  m
175                           else if (rAddr_Curr == 492)
176                              rNum_Next <= {1'b0, iData[35:30], 5'b00000};
177                           else if (rAddr_Curr == 493)
178                              rNum_Next <= {1'b0, iData[29:24], 5'b00000};
179                           else if (rAddr_Curr == 494)
180                              rNum_Next <= {1'b0, iData[23:18], 5'b00000};
181                           else if (rAddr_Curr == 495)
182                              rNum_Next <= {1'b0, iData[17:12], 5'b00000};
183                           else if (rAddr_Curr == 496)
184                              rNum_Next <= {1'b0, iData[11:6], 5'b00000};
185                           else if (rAddr_Curr == 497)
```

```
                                rNum_Next <= {1'b0, iData[5:0], 5'b00000};

                default   :   rNum_Next <= rNum_Curr;

    endcase
    end


    always @(posedge iClk)
    begin
        rAddr_Curr   <= rAddr_Next;
        rNum_Curr    <= rNum_Next;
    end

    assign oAddr = rAddr_Curr;
    assign oWe   = (rFSM_Curr == sIdle
            || rFSM_Curr == sInit
            || rFSM_Curr == sUpdate0
            || rFSM_Curr == sUpdate1
            || rFSM_Curr == sUpdate2
            || rFSM_Curr == sUpdate3
            || rFSM_Curr == sUpdate4
            || rFSM_Curr == sUpdate5
            || rFSM_Curr == sUpdate6
            || rFSM_Curr == sUpdate7
            || rFSM_Curr == sUpdate8
            || rFSM_Curr == sUpdate9
            || rFSM_Curr == sUpdate10) ? 1 : 0;
    assign oData = rNum_Curr;

endmodule
```

A.3 Python script for making the Static Screen file

Listing A.9: Static Screen code

```
from tkinter.filedialog import askopenfilename
from pathlib import Path

downloads_path = str(Path.home() / "Downloads")

lookuptable = {
    " ": 0,
    "!": 32,
    "\": 64,
    "#": 96,
    "$": 128,
    "%": 160,
    "&": 192,
    "\": 224,
    "(": 256,
    ")": 288,
    "*": 320,
    "+": 352,
    ",": 384,
    "-": 416,
    # Dot is 448, but to make it easier to make your initial file we put a dot
        for a space
    ".": 0,
    "/": 480,
    "0": 512,
    "1": 544,
    "2": 576,
```

```
    "3": 608,
    "4": 640,
    "5": 672,
    "6": 704,
    "7": 736,
    "8": 768,
    "9": 800,
    ":": 832,
    ";": 864,
    "<": 896,
    "=": 928,
    ">": 960,
    "?": 992,
    "@": 1024,
    "A": 1056,
    "B": 1088,
    "C": 1120,
    "D": 1152,
    "E": 1184,
    "F": 1216,
    "G": 1248,
    "H": 1280,
    "I": 1312,
    "J": 1344,
    "K": 1376,
    "L": 1408,
    "M": 1440,
    "N": 1472,
    "O": 1504,
    "P": 1536,
    "Q": 1568,
    "R": 1600,
    "S": 1632,
    "T": 1664,
    "U": 1696,
    "V": 1728,
    "W": 1760,
    "X": 1792,
    "Y": 1824,
    "Z": 1856,
    "[": 1888,
    "\\": 1920,
    "]": 1952,
    "^": 1984,
    "_": 2016,
    "\'": 2048,
    "a": 2080,
    "b": 2112,
    "c": 2144,
    "d": 2176,
    "e": 2208,
    "f": 2240,
    "g": 2272,
    "h": 2304,
    "i": 2336,
    "j": 2368,
    "k": 2400,
    "l": 2432,
    "m": 2464,
    "n": 2496,
    "o": 2528,
    "p": 2560,
    "q": 2592,
    "r": 2624,
    "s": 2656,
    "t": 2688,
    "u": 2720,
```

```
    "v": 2752,
    "w": 2784,
    "x": 2816,
    "y": 2848,
    "z": 2880,
    "{": 2912,
    "|": 2944,
    "}": 2976,
    "~": 3008
}


def makefile(hex_char):
    file1 = open(downloads_path + "/mystatic.txt", "w")
    file1.writelines(hex_char)
    file1.close()


def convert(file):
    # Convert file
    hex_char = []
    for i in range(len(file)):
        dec = lookuptable.get(file[i])
        # Remove 0x and add an enter after line
        hex_char.append(str(hex(dec)[2:] + "\n"))

    makefile(hex_char)


def choosefile():
    # Choose the file you want to convert
    filename = askopenfilename()
    file = open(filename, "r").read().replace("\n", "")
    print(file)

    convert(file)


if __name__ == '__main__':
    choosefile()
```

A.4 ScreenBufferMem module code

Listing A.10: ScreenBufferMem module code

```
`timescale 1ns / 1ps

module ScreenBufferMem #(
  parameter   WIDTH =  12,
  parameter   DEPTH =  600
  )
  (
  input   wire                          iClk,
  input   wire [$clog2(DEPTH)-1:0]      iAddrA, iAddrB,
  input   wire [WIDTH-1:0]              iDataB,
  input   wire                          iWeB,
  output  wire [WIDTH-1:0]              oDataA,oDataB
  );

  // define the memory
  reg [WIDTH-1:0] rMem  [DEPTH-1:0];

  // Initial contents of the memory
```

```
  initial
  begin
    $readmemh("static_mems.mem", rMem);
  end

  // Logic for Port A
  //  Supports only synchronous reading
  reg [WIDTH-1:0] rDataA;

  always @(posedge iClk)
  begin
    rDataA <= rMem[iAddrA];
  end

  assign oDataA = rDataA;

  // Logic for Port B
  //  Supports synchronous reading and writing
  reg [WIDTH-1:0] rDataB;

  always @(posedge iClk)
  begin
    if(iWeB)
      rMem[iAddrB] <= iDataB;
    rDataB <= rMem[iAddrB];
  end

  assign oDataB = rDataB;

endmodule
```