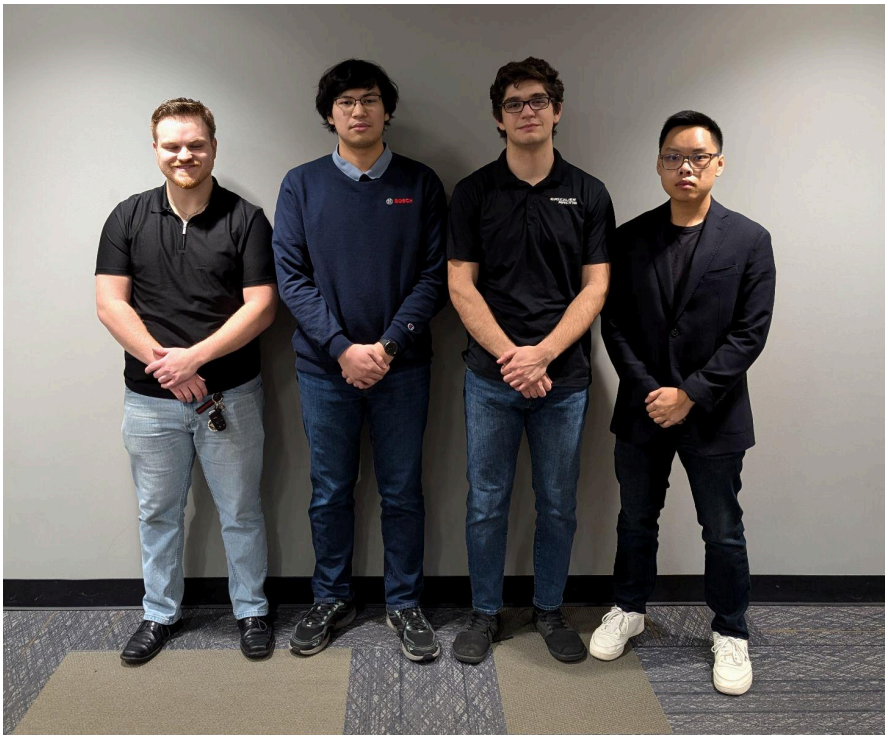
ECE/CSI 4710/5710 – Final Report

Professor Darrin Hanna



Group Number: 2

Lucas Costello Bao Nguyen Antonio Ristevski Ethan Vang

1. Abstract 3

2. Introduction 3

3. Top-Level Design 4

4. VHDL Design Details 14

4.1. PS/2 Keyboard Decoder 14

4.2. VGA Text Generator 17

4.3. RISC Processing Unit 25

5. Project Performance 34

7. References 37

8. Appendix 38

1. Abstract

This project takes on the creation of a RISC-inspired processing system with the Nexys A7-100T FPGA Trainer Board. The system consists of three primary components: a PS/2 keyboard decoder, a VGA display unit, and a RISC processor, which allows users to create and run assembly-like code shown on a VGA monitor. The project faced obstacles such as optimizing VGA rendering and fixing timing issues with the RISC processor. These concerns were addressed using iterative testing, resulting in a functional and cohesive system. Although the basic goals were met, there is room for improvement, such as the inclusion of an assembler within our processing unit and increased functionality of the VGA display. This project actively demonstrates the impressive capabilities of FPGA-based systems and shows how they can be utilized to work as prototypes for complex architectures like processors.

2. Introduction

The RISC (Reduced Instruction Set Computer) architecture was chosen for this project because of its simplicity and efficacy. RISC is well known for its fixed-length, 32-bit instructions that require two clock cycles to execute and for its predictable execution. These features make it ideal for processing data efficiently and troubleshooting more easily. This design also makes use of numerous accumulators, which lowers the frequency of memory access and boosts system performance. While designing this implementation of the processor, reference was made to the RV32I Base Integer Instruction Set [Waterman 21, 6] as a source of inspiration and key design elements.

In this project, we created a programming terminal and emulated a RISC processor so that users could develop, examine, and execute custom programs. Input and output are done via a PS/2 keyboard and a VGA display. A well-timed and adaptable architecture was required to ensure that the keyboard, processor, and display all worked in unison. This study delves into the system's architecture, implementation, and operation, outlining the obstacles faced and solutions identified along the route.

3. Top-Level Design

From a top down perspective, the system is composed of four major components: the PS/2 Keyboard Decoder, VGA Text Generator, RISC Processing Unit, and Seven-Segment Display. Using a keyboard, a user can write lines of code in hexadecimal that are displayed in real time on the VGA monitor. As the instructions are being written they are sent to the program ROM inside of the RISC processing unit. When the program is complete, it can be executed by the processing unit by flipping a switch on the FPGA board. After execution the contents of all 32 registers will be accessible on the seven-segment display, with register selection also available through on-board switches. Each component is crucial for the system's functionality and enables this design to emulate the terminal successfully.

The PS/2 Keyboard Decoder processes the user's input. This effectively converts the serial scan codes received from the keyboard, to ASCII characters that are used by other components. To do this, it employs a pulse generator for timing control and a shift register for serial-to-parallel data conversion. A lookup table is then utilized to translate the scan codes received from the keyboard into ASCII characters. After the conversion has occurred the corresponding ASCII is sent to both the VGA Text Generator and the ASCII Pulse Generator.

The pulse generator is responsible for transferring the individual ASCII characters to the RISC Processing Unit. The keyboard components were provided by Dr. Daniel Llamocca on his website [4].

The VGA Text Generator takes the ASCII characters and displays them on the monitor accordingly with a resolution of 640x480 pixels. Internally, this component houses all of the VGA display related logic. It utilizes RAM in the form of tile memory, having each character take up one character tile. All together the display is designed to allow for 80 maximum characters in width and 30 maximum characters in height. The text generator has built-in cursor management that increments the cursor to the right every time there's a character input. This is in addition to the function of going to the next line using the 'enter' key. This component directly outputs the horizontal and vertical synchronization signals, along with their respective RGB value, in order to create the textbox on the screen for the user.

The RISC Processing Unit stores and runs programs that are written by the user. Taking the ASCII input from the pulse generator, this unit internally stores the code character by character (in hexadecimal). For clarity, the contents of the program ROM mirror what is being displayed on the monitor for the user. Once the code is done being written, the user can activate the processing units clock by enabling switch 6 on the FPGA board. When the program is finished running, the contents of each register are output to a 32-to-1 mux that connects to the Seven-Segment Display. By using switches 4 through 0 on the FPGA board, the user can manually select which register's contents they want to be displayed.

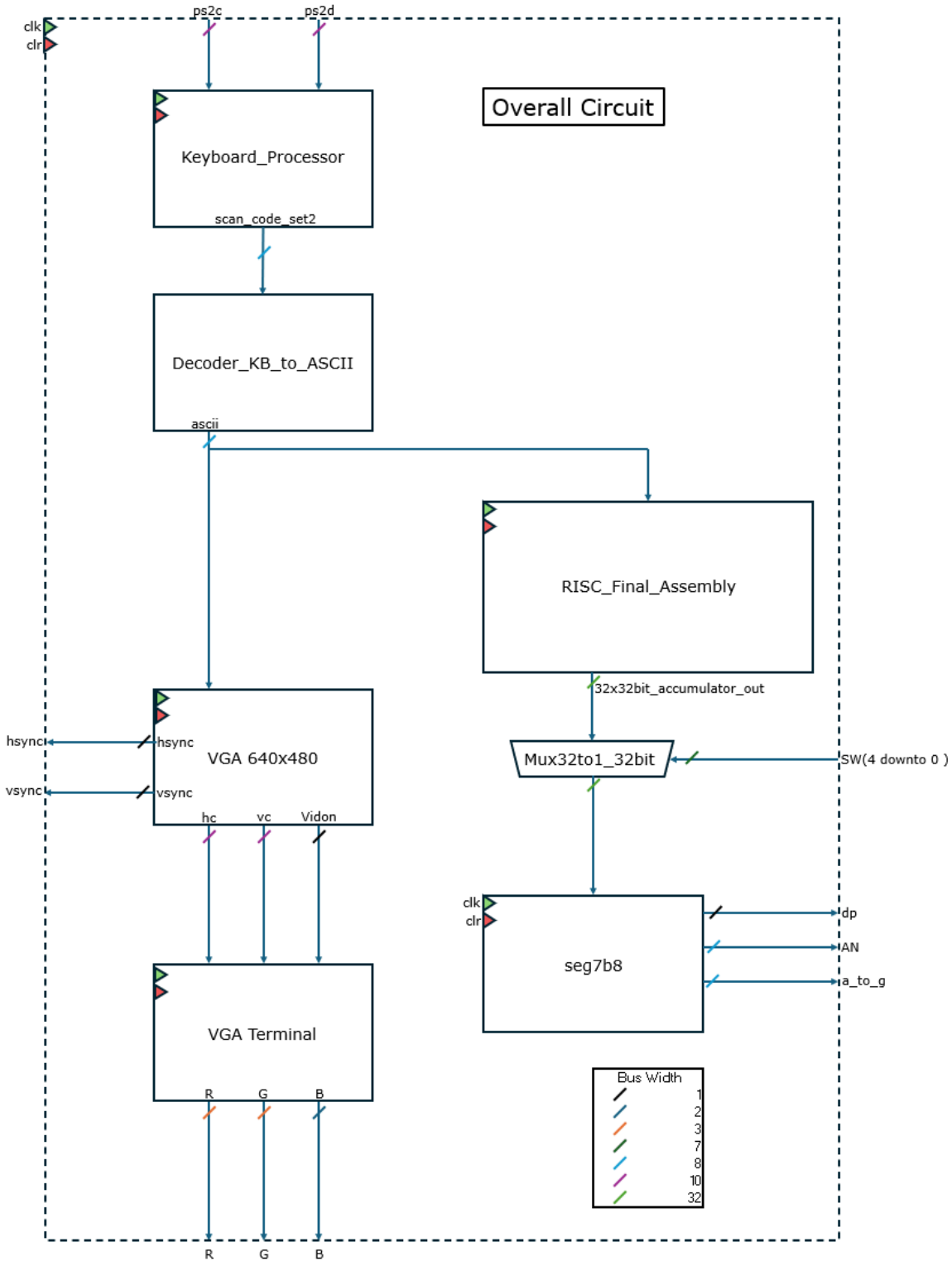


Figure 1: *terminal_final_assembly.vhd*

(Top-Level Schematic)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Terminal_Final_Assembly is
  Port (
        clk : in std_logic;
        rstn : in std_logic;
        ps2c : in std_logic;
        ps2d : in std_logic;
        SW : in std_logic_vector (6 downto 0);
        a_to_g : out std_logic_vector (6 downto 0);
        DP : out std_logic;
        AN : out std_logic_vector (7 downto 0);
        hsync : out std_logic;
        vsync : out std_logic;
        rgb : out std_logic_vector (2 downto 0)
        );
end Terminal_Final_Assembly;

architecture Behavioral of Terminal_Final_Assembly is

component clkdiv
    Port ( mclk : in STD_LOGIC;
           clr : in STD_LOGIC;
           clk25m : out STD_LOGIC;
           clk6m : out std_logic;
           clk3m : out std_logic
           );
end component;

component Keyboard_to_ASCII
           port (resetn, clock: in std_logic;
                                ps2c, ps2d: in std_logic;
                                DOUT: out std_logic_vector (7 downto 0);
                                done: out std_logic);
```

```
end component;

component ascii_pulse_generator
    Port ( clk : in STD_LOGIC;
           clr : in STD_LOGIC;
           ascii_in : in STD_LOGIC_VECTOR (7 downto 0);
           ascii_out_1_pulse : out std_logic_vector (7 downto 0);
           ascii_out_2_pulse : out STD_LOGIC_VECTOR (7 downto 0);
           keyboard_rstn : out std_logic
           );
end component;

component PROM_Programmer
    Port (
        clk : in STD_LOGIC;
        clr : in STD_LOGIC;
        ascii_in : in STD_LOGIC_VECTOR(7 downto 0);
        concadinated : out STD_LOGIC_VECTOR(63 downto 0);
        done : out STD_LOGIC
    );
end component;

component concadinated_to_hex
    Port (
        concadinated : in STD_LOGIC_VECTOR(63 downto 0);
        hex : out STD_LOGIC_VECTOR(31 downto 0)
    );
end component;

component RISC_Final_Assembly
  Port (
        clk : in std_logic;
        clr : in std_logic;
        ascii : in std_logic_vector (7 downto 0);
        processor_load : in std_logic;
        X0_out : out std_logic_vector (31 downto 0);
        X1_out : out std_logic_vector (31 downto 0);
        X2_out : out std_logic_vector (31 downto 0);
        X3_out : out std_logic_vector (31 downto 0);
        X4_out : out std_logic_vector (31 downto 0);
        X5_out : out std_logic_vector (31 downto 0);
        X6_out : out std_logic_vector (31 downto 0);
        X7_out : out std_logic_vector (31 downto 0);
        X8_out : out std_logic_vector (31 downto 0);
        X9_out : out std_logic_vector (31 downto 0);
        X10_out : out std_logic_vector (31 downto 0);
```

```
        X11_out : out std_logic_vector (31 downto 0);
        X12_out : out std_logic_vector (31 downto 0);
        X13_out : out std_logic_vector (31 downto 0);
        X14_out : out std_logic_vector (31 downto 0);
        X15_out : out std_logic_vector (31 downto 0);
        X16_out : out std_logic_vector (31 downto 0);
        X17_out : out std_logic_vector (31 downto 0);
        X18_out : out std_logic_vector (31 downto 0);
        X19_out : out std_logic_vector (31 downto 0);
        X20_out : out std_logic_vector (31 downto 0);
        X21_out : out std_logic_vector (31 downto 0);
        X22_out : out std_logic_vector (31 downto 0);
        X23_out : out std_logic_vector (31 downto 0);
        X24_out : out std_logic_vector (31 downto 0);
        X25_out : out std_logic_vector (31 downto 0);
        X26_out : out std_logic_vector (31 downto 0);
        X27_out : out std_logic_vector (31 downto 0);
        X28_out : out std_logic_vector (31 downto 0);
        X29_out : out std_logic_vector (31 downto 0);
        X30_out : out std_logic_vector (31 downto 0);
        X31_out : out std_logic_vector (31 downto 0)
         );
end component;

component mux32to1_32bit
  Port (
        a : in std_logic_vector (31 downto 0); -- 1
        b : in std_logic_vector (31 downto 0); -- 2
        c : in std_logic_vector (31 downto 0); -- 3
        d : in std_logic_vector (31 downto 0); -- 4
        e : in std_logic_vector (31 downto 0); -- 5
        f : in std_logic_vector (31 downto 0); -- 6
        g : in std_logic_vector (31 downto 0); -- 7
        h : in std_logic_vector (31 downto 0); -- 8
        i : in std_logic_vector (31 downto 0); -- 9
        j : in std_logic_vector (31 downto 0); -- 10
        k : in std_logic_vector (31 downto 0); -- 11
        l : in std_logic_vector (31 downto 0); -- 12
        m : in std_logic_vector (31 downto 0); -- 13
        n : in std_logic_vector (31 downto 0); -- 14
        o : in std_logic_vector (31 downto 0); -- 15
        p : in std_logic_vector (31 downto 0); -- 16
        q : in std_logic_vector (31 downto 0); -- 17
        r : in std_logic_vector (31 downto 0); -- 18
        s : in std_logic_vector (31 downto 0); -- 19
        t : in std_logic_vector (31 downto 0); -- 20
```

```
        u : in std_logic_vector (31 downto 0); -- 21
        v : in std_logic_vector (31 downto 0); -- 22
        w : in std_logic_vector (31 downto 0); -- 23
        x : in std_logic_vector (31 downto 0); -- 24
        y : in std_logic_vector (31 downto 0); -- 25
        z : in std_logic_vector (31 downto 0); -- 26
        aa : in std_logic_vector (31 downto 0); -- 27
        bb : in std_logic_vector (31 downto 0); -- 28
        cc : in std_logic_vector (31 downto 0); -- 29
        dd : in std_logic_vector (31 downto 0); -- 30
        ee : in std_logic_vector (31 downto 0); -- 31
        ff : in std_logic_vector (31 downto 0); -- 32
        sel : in std_logic_vector (4 downto 0);
        zout : out std_logic_vector (31 downto 0)
        );
end component;

component text_generation_top
  Port (
        clk : in std_logic;
        rstn : in std_logic;
        btn : in std_logic_vector (2 downto 0);
        sw : in std_logic_vector (6 downto 0);
        hsync, vsync : out std_logic;
        rgb : out std_logic_vector (2 downto 0);
        keyboard_rstn : out std_logic;
        exec_code : out std_logic
        );
end component;

component x7segb8
            port(
                        x : in STD_LOGIC_VECTOR(31 downto 0);
                        clk : in STD_LOGIC;
                        clr : in STD_LOGIC;
                        a_to_g : out STD_LOGIC_VECTOR(6 downto 0);
                        an : out STD_LOGIC_VECTOR(7 downto 0);
                        dp : out STD_LOGIC
                );
end component;

signal clr, clk6m, keyboard_rstn, keyboard_rstn_sig, keyboard_vga : std_logic;
signal ascii, ascii_pulse, ascii_2_pulse : std_logic_vector (7 downto 0);
signal  X0_out, X1_out, X2_out, X3_out, X4_out, X5_out, X6_out, X7_out, X8_out, X9_out,
        X10_out, X11_out, X12_out, X13_out, X14_out, X15_out, X16_out, X17_out, X18_out,
X19_out, X20_out, X21_out, X22_out, X23_out, X24_out, X25_out, X26_out, X27_out,
```

```
X28_out, X29_out, X30_out, X31_out, seg7_in : std_logic_vector (31 downto 0);

begin

clr <= not(rstn);

clkdivider : clkdiv port map    (mclk => clk,
                          clr => clr,
                          clk25m => open,
                          clk6m => clk6m,
                          clk3m => open);

keyboard_rstn <= rstn and keyboard_vga;

keyboard : Keyboard_to_ASCII port map   (clock => clk,
                                resetn => keyboard_rstn,
                                ps2c => ps2c,
                                ps2d => ps2d,
                                dout => ascii,
                                done => open);

pulsegen : ascii_pulse_generator port map   (clk => clk,
                                   clr => clr,
                                   ascii_in => ascii,
                                   ascii_out_1_pulse => ascii_pulse,
                                   ascii_out_2_pulse => ascii_2_pulse,
                                   keyboard_rstn => keyboard_rstn_sig);

RISC_PM : RISC_Final_Assembly port map  (clk => clk,
                                clr => clr,
                                ascii => ascii_pulse,
                                processor_load => SW(6),
                                X0_out => X0_out,
                                X1_out => X1_out,
                                X2_out => X2_out,
                                X3_out => X3_out,
                                X4_out => X4_out,
                                X5_out => X5_out,
                                X6_out => X6_out,
                                X7_out => X7_out,
                                X8_out => X8_out,
                                X9_out => X9_out,
                                X10_out => X10_out,
                                X11_out => X11_out,
                                X12_out => X12_out,
                                X13_out => X13_out,
```

```
                                X14_out => X14_out,
                                X15_out => X15_out,
                                X16_out => X16_out,
                                X17_out => X17_out,
                                X18_out => X18_out,
                                X19_out => X19_out,
                                X20_out => X20_out,
                                X21_out => X21_out,
                                X22_out => X22_out,
                                X23_out => X23_out,
                                X24_out => X24_out,
                                X25_out => X25_out,
                                X26_out => X26_out,
                                X27_out => X27_out,
                                X28_out => X28_out,
                                X29_out => X29_out,
                                X30_out => X30_out,
                                X31_out => X31_out);

accum_mux : mux32to1_32bit port map (   a => X0_out,
                                b => X1_out,
                                c => X2_out,
                                d => X3_out,
                                e => X4_out,
                                f => X5_out,
                                g => X6_out,
                                h => X7_out,
                                i => X8_out,
                                j => X9_out,
                                k => X10_out,
                                l => X11_out,
                                m => X12_out,
                                n => X13_out,
                                o => X14_out,
                                p => X15_out,
                                q => X16_out,
                                r => X17_out,
                                s => X18_out,
                                t => X19_out,
                                u => X20_out,
                                v => X21_out,
                                w => X22_out,
                                x => X23_out,
                                y => X24_out,
                                z => X25_out,
                                aa => X26_out,
```

```
                              bb => X27_out,
                              cc => X28_out,
                              dd => X29_out,
                              ee => X30_out,
                              ff => X31_out,
                              sel => SW(4 downto 0),
                              zout => seg7_in
                                );

seg7 : x7segb8 port map (clk => clk,
                    clr => clr,
                    x => seg7_in,
                    a_to_g => a_to_g,
                    dp => dp,
                    an => an);

vga_textbox : text_generation_top port map  (clk => clk,
                                    rstn => rstn,
                                    btn => "000",
                                    sw => ascii(6 downto 0),
                                    hsync => hsync,
                                    vsync => vsync,
                                    rgb => rgb,
                                    keyboard_rstn => keyboard_vga,
                                    exec_code => open);


end Behavioral;
```

Code 1: *Terminal_Final_Assembly.vhd*

(The VHDL code of the terminal's final assembly)

4. VHDL Design Details

This section will discuss the three main components of the project in detail: PS/2 Keyboard Decoder, VGA Textbox, and RISC Processor.

4.1. PS/2 Keyboard Decoder

The PS/2 to ASCII Decoder is one of the crucial components of the system as it enables users to interact with the project through a PS/2 keyboard. This component translates raw scan codes, that are generated by the keyboard, into standard ASCII codes that are used by other components in the system. It does this by capturing data from the keyboard and filtering it to ensure there was a valid keypress. If the keypress was valid, the scan code is decoded into its corresponding ASCII equivalent.

Within this component lies the main module: my_ps2keyboard. This module interfaces with the PS/2 clock and data lines (ps2c, ps2d) to capture keyboard signals. As the signals are captured, they are read and filtered to determine validity. Once a valid scan code is captured by the reader it outputs the 8-bit scan code along with a done signal. The purpose of the done signal is to ensure that the data is only transferred if it's high, meaning that any scan codes that are invalid get blocked.

After a valid scan code is output from my_ps2keyboard it is sent to the Scan_Code_to_ASCII_Decoder. Just as the name suggests, this decoder uses a lookup table to map the validated scan codes to their corresponding ASCII values whether they are characters, numbers, or special keys. The ASCII output from the decoder is sent to other modules as a one clock cycle pulse, after which the keyboard resets and waits for the next validated keypress.

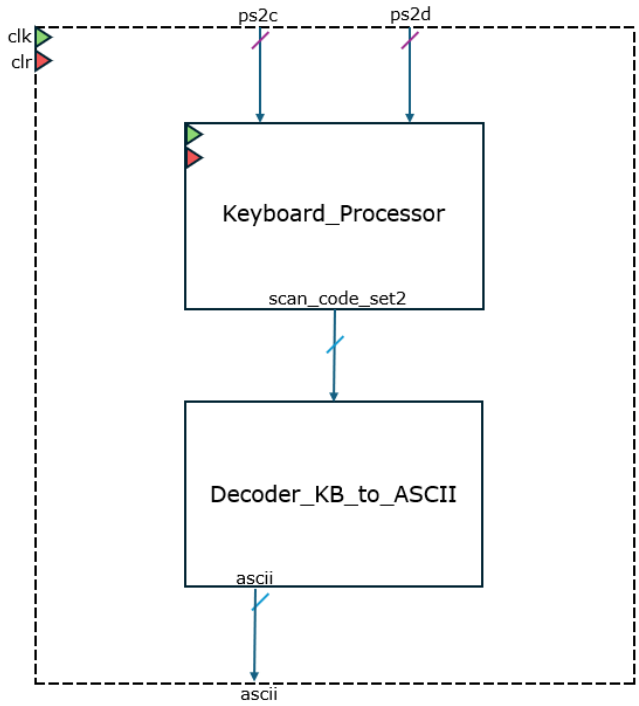


Figure 2: *Keyboard_to_ASCII.vhd*

(PS/2 Keyboard Decoder into ASCII)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.ASCII_Table.all;

entity ScanCode_to_ASCII_Decoder is
    Port (
        ScanCode : in STD_LOGIC_VECTOR (7 downto 0);
        ASCII    : out STD_LOGIC_VECTOR (7 downto 0)
    );
end ScanCode_to_ASCII_Decoder;

architecture Behavioral of ScanCode_to_ASCII_Decoder is
begin
    ASCII <= UpperA_ASCII when ScanCode = x"1C" else
             UpperB_ASCII when ScanCode = x"32" else
             UpperC_ASCII when ScanCode = x"21" else
             UpperD_ASCII when ScanCode = x"23" else
             UpperE_ASCII when ScanCode = x"24" else
             UpperF_ASCII when ScanCode = x"2B" else
```

```
        UpperG_ASCII when ScanCode = x"34" else
        UpperH_ASCII when ScanCode = x"33" else
        UpperI_ASCII when ScanCode = x"43" else
        UpperJ_ASCII when ScanCode = x"3B" else
        UpperK_ASCII when ScanCode = x"42" else
        UpperL_ASCII when ScanCode = x"4B" else
        UpperM_ASCII when ScanCode = x"3A" else
        UpperN_ASCII when ScanCode = x"31" else
        UpperO_ASCII when ScanCode = x"44" else
        UpperP_ASCII when ScanCode = x"4D" else
        UpperQ_ASCII when ScanCode = x"15" else
        UpperR_ASCII when ScanCode = x"2D" else
        UpperS_ASCII when ScanCode = x"1B" else
        UpperT_ASCII when ScanCode = x"2C" else
        UpperU_ASCII when ScanCode = x"3C" else
        UpperV_ASCII when ScanCode = x"2A" else
        UpperW_ASCII when ScanCode = x"1D" else
        UpperX_ASCII when ScanCode = x"22" else
        UpperY_ASCII when ScanCode = x"35" else
        UpperZ_ASCII when ScanCode = x"1A" else
        Num0_ASCII when ScanCode = x"45" else
        Num1_ASCII when ScanCode = x"16" else
        Num2_ASCII when ScanCode = x"1E" else
        Num3_ASCII when ScanCode = x"26" else
        Num4_ASCII when ScanCode = x"25" else
        Num5_ASCII when ScanCode = x"2E" else
        Num6_ASCII when ScanCode = x"36" else
        Num7_ASCII when ScanCode = x"3D" else
        Num8_ASCII when ScanCode = x"3E" else
        Num9_ASCII when ScanCode = x"46" else
        Space when ScanCode = x"29" else
        Carriage_Return when ScanCode = x"5A" else
        Escape when ScanCode = x"76" else
                        Comma_ASCII when ScanCode = x"41" else
                        Hashtag_ASCII when ScanCode = x"04" else
                        Address_ASCII when ScanCode = x"06" else
        (others => '0');
end Behavioral;
```

Code 2: *Decoder_KB_to_ASCII.vhd*

(The decoder that converts the keyboard scan codes into ASCII)

4.2. VGA Text Generator

All of the VGA-related aspects of this project are handled by the VGA_Text_Generator_top. The clock divider module, labelled clkdiv.vhd, divides the FPGA's 100 MHz clock to 25 MHz. This satisfies the VGA speed requirement, allowing the monitor to display properly. The vga_640x480 module generates the timing signals 'hsync' and 'vsync' that instruct the monitor on when to begin and end displaying each line and frame. This guarantees that the screen will display everything accurately.

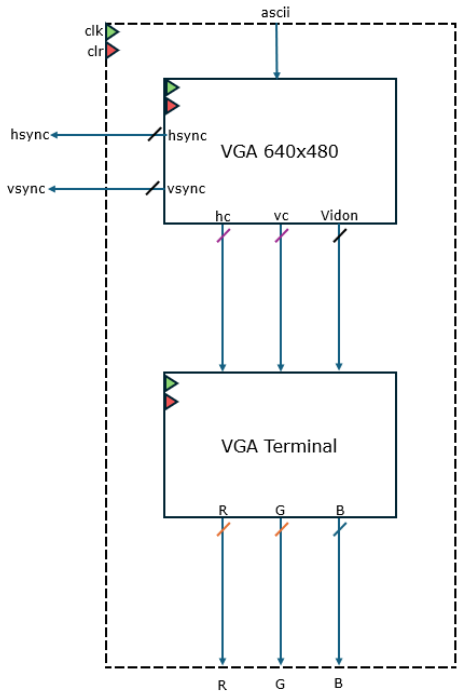


Figure 3: *VGA_Text_Generator_top.vhd*

(Top-Level Schematic for the VGA Text Generator)

The task of transforming the ASCII character input into readable text falls to the text_screen_gen module. The module manages both input character storage and serves as the system's main text renderer. This module interfaces with the tile memory, the ROM that stores

the font characters, and VGA display. These work together to produce the pixel-based character representations that are shown on the screen. Each ASCII characters' 8x16 bitmap is stored in the font ROM, and the active bitmap of the character that will be shown on the screen is kept in the tile memory. The ASCII values corresponding to each character tile are retrieved from the tile memory as the VGA's horizontal and vertical counters record the screen's refresh cycle. These ASCII values get sent through the font ROM to be converted into pixel data for display.

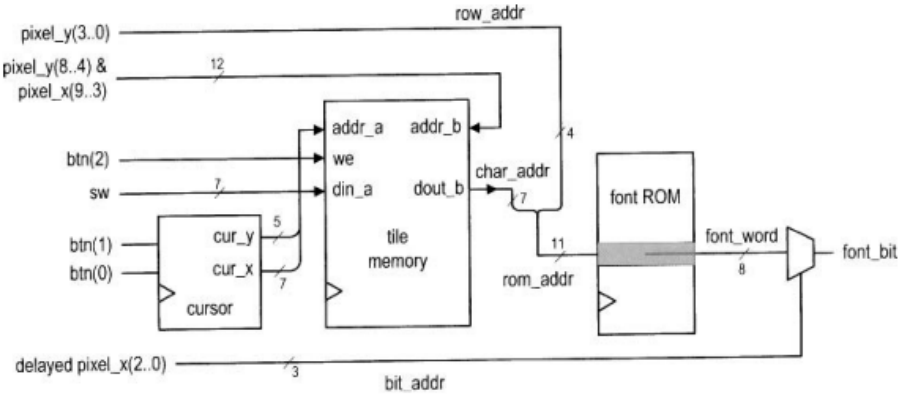


Figure 4: *VGA Text Generation Circuit using Tile Memory [Chu 298, 2]*

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use IEEE.std_logic_unsigned.all;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity text_screen_gen is
    Port ( clk : in STD_LOGIC;
```

```
        clr : in STD_LOGIC;
        btn : in STD_LOGIC_VECTOR (2 downto 0);
        ascii : in STD_LOGIC_VECTOR (6 downto 0);
        vidon : in STD_LOGIC;
        hc : in STD_LOGIC_VECTOR (9 downto 0);
        vc : in STD_LOGIC_VECTOR (9 downto 0);
        text_rgb : out STD_LOGIC_VECTOR (2 downto 0));
end text_screen_gen;

architecture Behavioral of text_screen_gen is

component pulse_generator
    Port ( clk : in STD_LOGIC;
        clr : in STD_LOGIC;
        d : in STD_LOGIC;
        q : out STD_LOGIC);
end component;

component fonts
  PORT (
    clka : IN STD_LOGIC;
    addra : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END component;

component tile_memory4096x8
  PORT (
    a : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
    d : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    dpra : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
    clk : IN STD_LOGIC;
    we : IN STD_LOGIC;
    dpo : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END component;

component block_tile_memory_4096x8
  PORT (
    clka : IN STD_LOGIC;
    wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    addra : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
    dina : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    clkb : IN STD_LOGIC;
    addrb : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
    doutb : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
```

```
  );
END component;

component nbitregister
  generic(N:integer := 4);
  Port (load : in std_logic;
        clk : in std_logic;
        clr : in std_logic;
        d : in std_logic_vector (N-1 downto 0);
        q : out std_logic_vector (N-1 downto 0));
end component;

    -- font ROM
    signal char_addr : std_logic_vector (6 downto 0);
    signal rom_addr : std_logic_vector (10 downto 0);
    signal row_addr : std_logic_vector (3 downto 0);
    signal bit_addr : std_logic_vector (2 downto 0);
    signal font_word : std_logic_vector (7 downto 0);
    signal font_bit : std_logic;
    -- tile memory
    signal we : std_logic;
    signal addr_r, addr_w : std_logic_vector (11 downto 0);
    signal din, dout : std_logic_vector (6 downto 0);
    -- 80x30 character map
    constant hc_max : integer := 80;
    constant vc_max : integer := 30;
    -- cursor
    signal cur_hc_reg, cur_hc_next : std_logic_vector (6 downto 0);
    signal cur_vc_reg, cur_vc_next : std_logic_vector (4 downto 0);
    signal btn0_tick, btn1_tick: std_logic;
    signal cursor_on : std_logic;
    -- delayed hc and vc
    signal hc1_reg, vc1_reg : std_logic_vector (9 downto 0);
    signal hc2_reg, vc2_reg : std_logic_vector (9 downto 0);
    -- object output signals
    signal font_rgb, font_rev_rgb : std_logic_vector (2 downto 0);

    -- cursor intermediate signals
    signal dout_sig : std_logic_vector (7 downto 0);
    signal wea : std_logic_vector (0 downto 0);

begin

btn0pulse : pulse_generator port map    (clk => clk,
                                clr => clr,
                                d => btn(0),
```

```
                                q => btn0_tick);

btn1pulse : pulse_generator port map    (clk => clk,
                                clr => clr,
                                d => btn(1),
                                q => btn1_tick);

fontrom : fonts port map    (clka => clk,
                        addra => '0' & rom_addr,
                        douta => font_word);

tileram : tile_memory4096x8 port map    (a => addr_w,
                                d => '0' & din,
                                dpra => addr_r,
                                clk => clk,
                                we => we,
                                dpo => dout_sig);

dout <= dout_sig (6 downto 0);

cursorx : nbitregister generic map  (N => 7)
                    port map    (clk => clk,
                                clr => clr,
                                load => '1',
                                d => cur_hc_next,
                                q => cur_hc_reg);

cursory : nbitregister generic map  (N => 5)
                    port map    (clk => clk,
                                clr => clr,
                                load => '1',
                                d => cur_vc_next,
                                q => cur_vc_reg);

hc1_delay : nbitregister generic map    (N => 10)
                        port map    (clk => clk,
                                clr => clr,
                                load => '1',
                                d => hc,
                                q => hc1_reg);

hc2_delay : nbitregister generic map    (N => 10)
                        port map    (clk => clk,
                                clr => clr,
                                load => '1',
                                d => hc1_reg,
```

```
                        q => hc2_reg);

vc1_delay : nbitregister generic map    (N => 10)
                port map    (clk => clk,
                        clr => clr,
                        load => '1',
                        d => vc,
                        q => vc1_reg);

vc2_delay : nbitregister generic map    (N => 10)
                port map    (clk => clk,
                        clr => clr,
                        load => '1',
                        d => vc1_reg,
                        q => vc2_reg);

-- tile memory write
addr_w <= cur_vc_reg & cur_hc_reg;
we <= btn(2);
din <= ascii;

-- tile mrmory read
addr_r <= vc(8 downto 4) & hc(9 downto 3);
char_addr <= dout;

-- font ROM
row_addr <= vc(3 downto 0);
rom_addr <= char_addr & row_addr;

bit_addr <= hc2_reg(2 downto 0);
font_bit <= font_word(to_integer(unsigned((not bit_addr))));

-- new cursor position
--cur_hc_next <=  (others => '0') when btn0_tick = '1' and cur_hc_reg = (hc_max - 1);
cur_hc_next <=  (others => '0') when btn0_tick = '1' and cur_hc_reg = "1001111" else
            cur_hc_reg + 1 when btn0_tick = '1' else
            cur_hc_reg;
cur_vc_next <=  (others => '0') when btn1_tick = '1' and cur_vc_reg = "11101" else
            cur_vc_reg + 1 when btn1_tick = '1' else
            cur_vc_reg;

-- green over black and reversed for video for cursor, I have no idea what this means
font_rgb <= "111" when font_bit = '1' else
          "000";
font_rev_rgb <= "000" when font_bit = '1' else
          "111";
```

```
cursor_on <= '1' when vc2_reg(8 downto 4) = cur_vc_reg and hc2_reg(9 downto 3) =
cur_hc_reg else
          '0';

-- RGB mux circuit
process(vidon, cursor_on, font_rgb, font_rev_rgb)
begin

   if vidon = '0' then
      text_rgb <= "000"; -- blank
   else
      if cursor_on = '1' then
         text_rgb <= font_rev_rgb;
      else
         text_rgb <= font_rgb;
      end if;
   end if;

end process;

end Behavioral;
```

Code 3: *VGA_Terminal.vhd*

(Implementation of the VGA textbox schematic in VHDL)

Apart from text rendering, the VGA Text Generator also manages the cursor. The integrated cursor management system has the capability to increment the cursor to the right or downwards. To properly emulate a text editor, the cursor should increment to the right for every key pressed on the keyboard, while also having the capability to increment downwards and return back to the starting position when enter is pressed. Given the limitation presented in the primitive cursor control method and the inclusion of VRAM, it presented a challenge with controlling the text editor on the screen. Thus, an FSM for the VGA textbox was designed. When the FSM initializes, it will increment through each pixel on the 640x480 screen to clear the pixels. Afterwards, it will position the cursor in the starting position for the first line in the text

box. When receiving an ASCII input, it will display the value in the cursor's position, then increment to the right. When the FSM receives the ASCII input of enter, it will increment downwards, while also incrementing to the right to loop back to the starting position of that line.

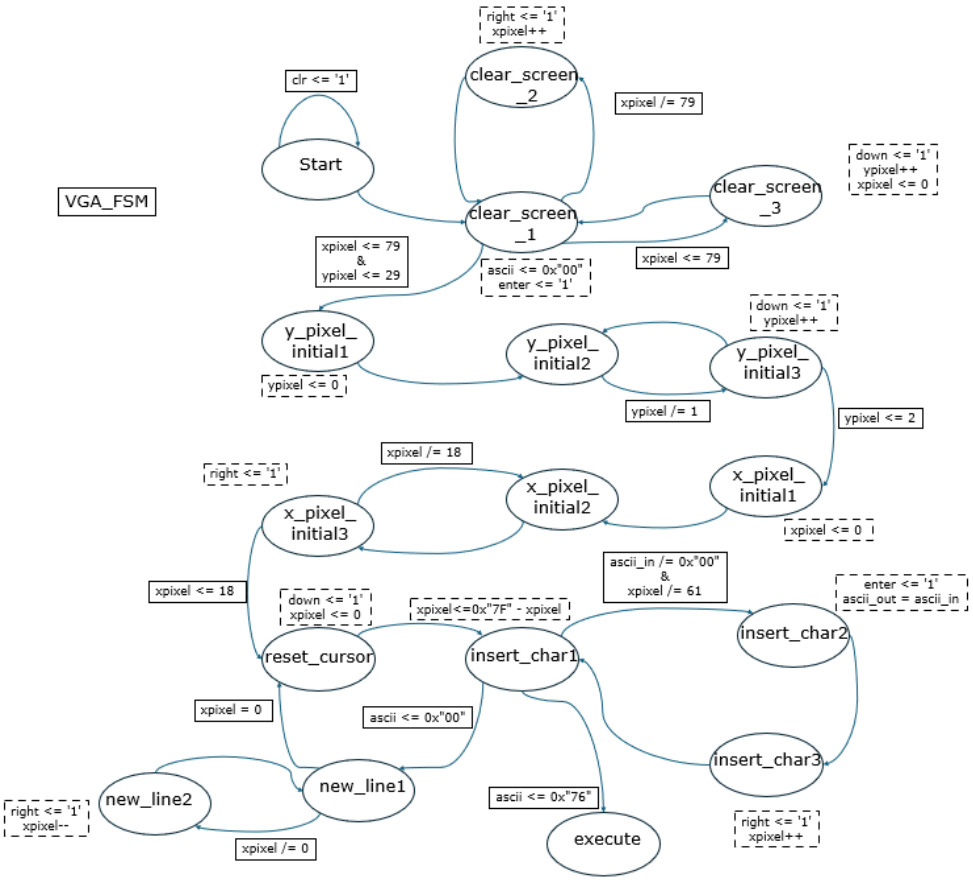


Figure 5: *VGA_text_screen_FSM.vhd*

(State machine that manages the textbox)

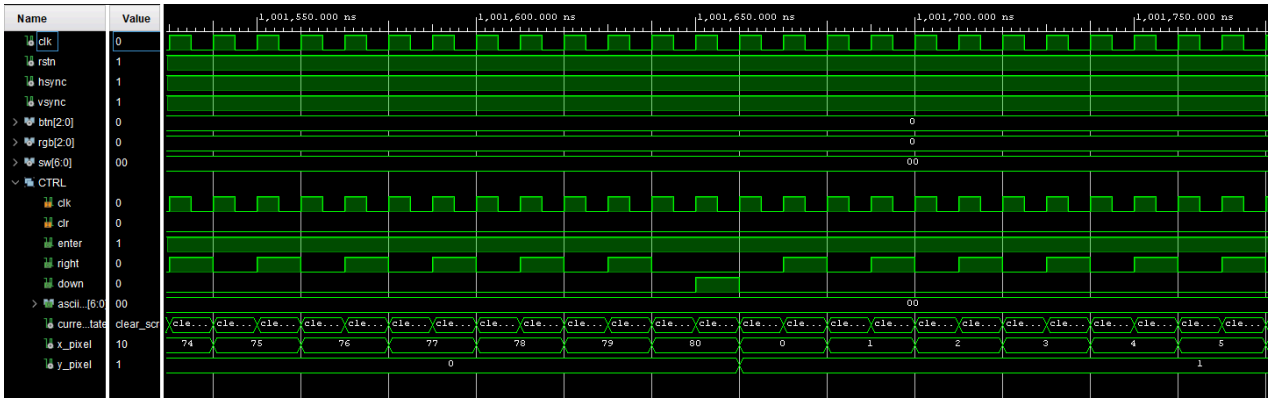


Figure 6: *VGA_text_screen_FSM.vhd*

(Simulation of the clear screen initialization)

4.3. RISC Processing Unit

The RISC Processing Unit consists of several essential components that work seamlessly together to run the program written by the user and output the results. Using an ASCII-based interface, users write their programs in hexadecimal that correspond to the specific 32-bit machine code. The RISC processor has four different instruction types: R-type, I-type, S-type, and B-type. As illustrated in ***Figure 4***, different instruction types utilize the 32-bit machine code differently. R-type is used for arithmetic and boolean expressions between 2 registers, where the output is stored in another register. I-type is used for arithmetic and boolean expressions between a register and immediate value, where the output is stored in another register. S-type is used in memory access operations where the value inside of an accumulator is stored into a memory address. Finally, B-type is used in branching operations, where the internal program counter will update from the given value in the instruction.

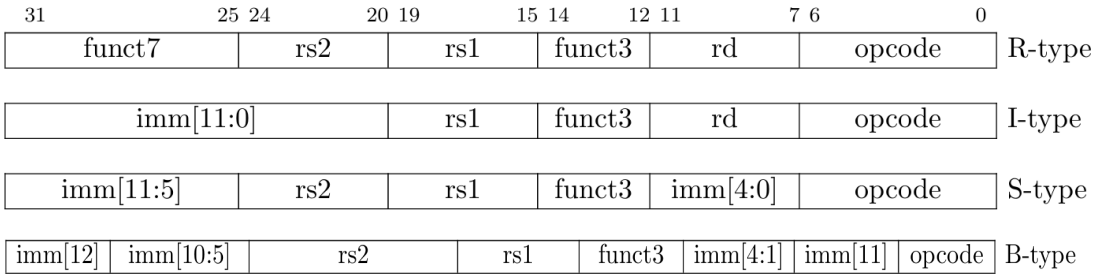


Figure 7: *RISC Instruction Type & Machine Code Format (Waterman and Asanović 12)*

In the instruction type tables, “opcode” represents the unique opcode for each operation, “Rd” represents the register destination, “Rs1” represents the first operand in an operation, “Rs2” represents the second operand in an operation, and “imm” represents an immediate value or memory location. The “funct” commands are used as bit extensions of the machine code into 32 bit. A list of all implemented opcodes and their associated instruction type is located in the appendix. The computation of each machine code operation starts with identifying the type of instruction, which is determined by the opcode assigned to the operation. Next is identifying what register addresses will be used in the instruction. Last is determining the immediate value used in the instruction. For example, to convert the instruction “XORI X14,X15,#A3D” starts with identifying the instruction type of opcode, which is an I type. Next, the opcode for XORI is “0001100.” Then, the accumulator addresses are converted into binary. Finally is inputting the immediate value from the instruction. At the end, the machine code in hexadecimal is found to be “A3D7870C.”

Immediate												Rs				Misc			Rd				Opcode							
1	0	1	0	0	0	1	1	1	1	0	1	0	1	1	1	1	0	0	0	0	1	1	1	0	0	0	0	1	1	0	0	XORI X14,X15,#A3D
A				3				D				7				8			7				0				C			

Figure 8: *Bitwise conversion from assembly to machine code*

Originally, an assembler was made that would allow the user to program in assembly language. Afterwards, the assembler would convert the written language and convert it into machine code. The assembler was fully designed, simulated, programmed, and implemented for the project. The assembler was designed to utilize a queue, or a first in first out storage method. This would allow the ASCII inputs from the keyboard to be stored while the user was typing out the code. After the enter key would be pressed, the assembler would dequeue the first four ASCII characters and load them into a register. This register was responsible for temporarily storing the first four characters. The importance of the first four characters is that every instruction made in this instruction set has exactly four characters, thus making the process of designing the assembler much easier. From there, the assembler would look for certain symbols and spacing in order to compile the rest of the language. Going back to the "XORI X14,X15,#A3D" example, the assembler would first load the X, O, R, and I. From there, it would determine that this instruction is an i type, where it would proceed to search for two digits after an X to designate the register destination, then search for the next two digits after another X to mark the register operand, and finally search for the final three numbers after the hashtag to mark where the immediate value is. **Figure 9 - Figure 10** will respectively display the assembler's top level design, FSM, and simulation.

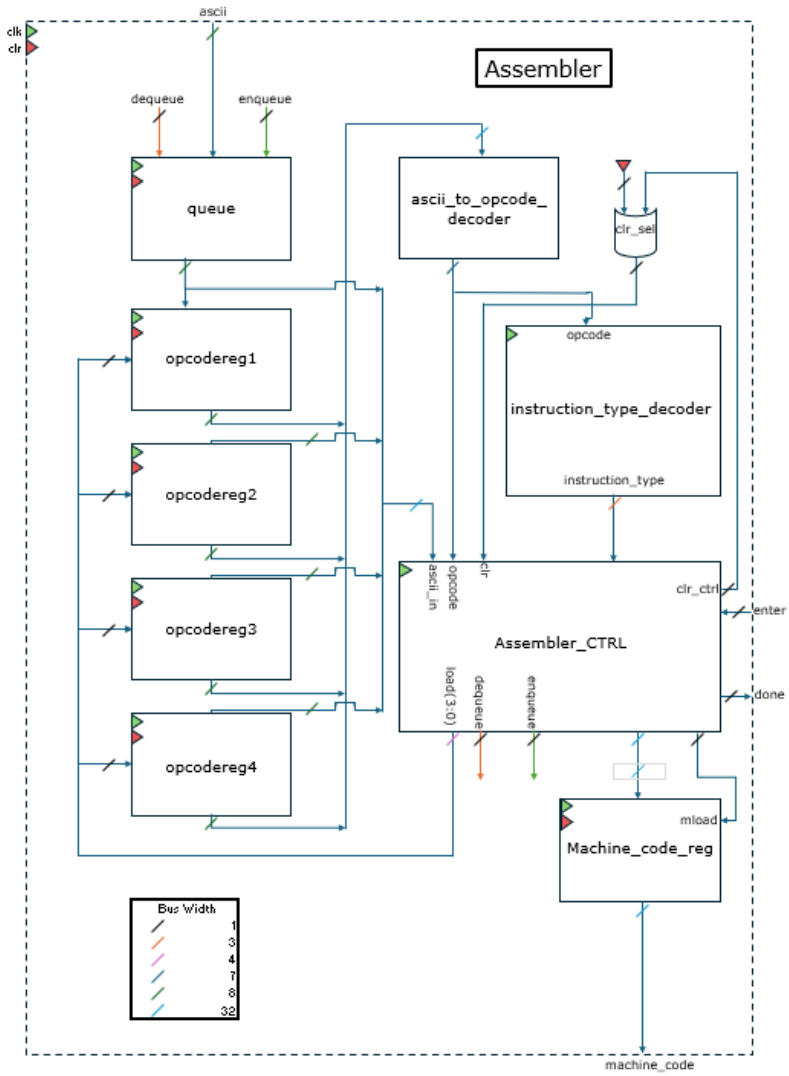


Figure 9: *Assembler_RISC.vhd*

(Top-level schematic of the designed RISC assembler)

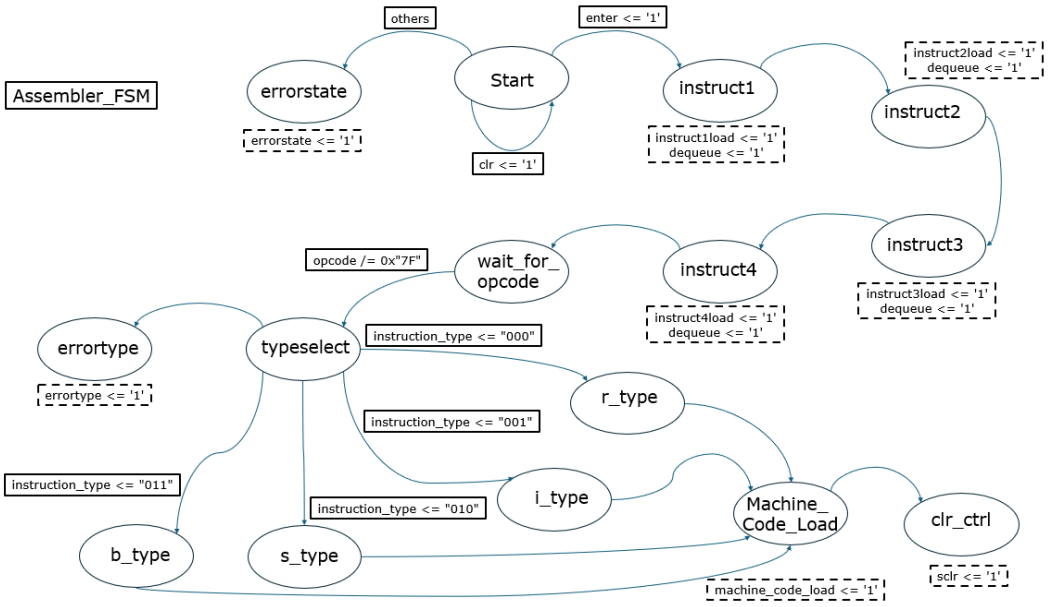


Figure 10: *Assember_CTRL.vhd*

(A state machine flow chart of the assembler)

In the simulations, the assembler worked as intended, displaying the correct machine code from the testbench. However, when the assembler was implemented onto the board for the final check of compatibility, the assembler would not output the correct values. It was decided due to the lack of time and debugging tools, the decision was made to leave the assembler behind and prioritize other aspects of the project. The simulation can be found in **Figure 11**.

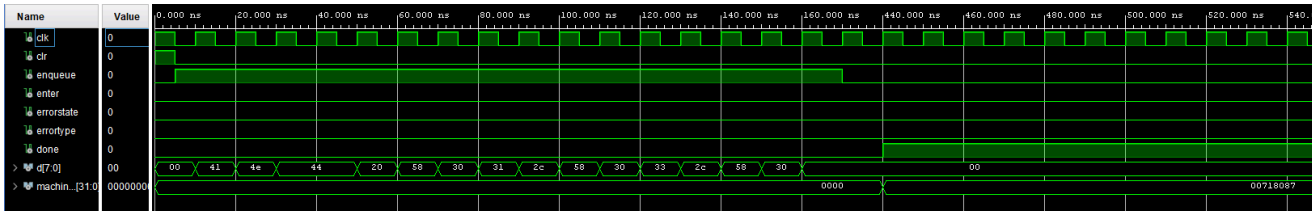


Figure 11: *Simulation of the assembler*

Outside of the RISC processor is a Tri-State buffer that controls the clock signal for the processor, enabling or disabling program execution as needed. When the Tri-State buffer is deactivated, the PROM is set up to be written to by the PROM Programmer. When the Tri-State buffer is enabled, the processor will read each instruction from the PROM and execute accordingly. Each instruction in the PROM executes for 2 clock cycles, which was found ideal to allow the accumulators, registers, and memory to stay in sync. After the program has completed its execution, the results are output to their respective registers and made available for viewing on the seven-segment display. It should be noted that accumulator 0 is used as the ZSR, or the accumulator whose value will always be zero. This was done for simplicity in programming the processor due to the default value being zeros in the machine code. The seven-segment display will display one accumulator at a time, where switches 0 to 5 will allow the user to select between the 32 accumulators.

The PROM Programmer is the medium between the user's input and the processor. As the user is typing onto the VGA textbox, the module will take the ASCII characters and concatenate them into a raw 64-bit output. From there, the 64-bit ASCII output is decoded into a 32-bit hexadecimal value, properly formatting the instruction to be suitable for machine code. The 32-bit machine code is then stored within the PROM to prepare for execution. This process takes effect everytime the user presses enter, essentially writing to the PROM in real-time as the user programs.

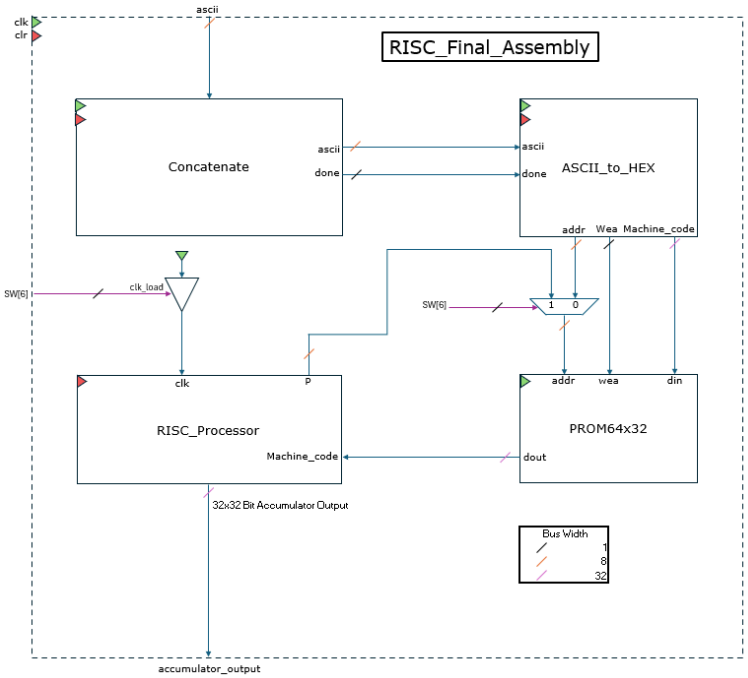


Figure 9: *RISC_Final_Assembly.vhd*

(Top-Level Schematic for RISC)

At the center of this system is the RISC Processor Core, which executes the machine code stored in the PROM. The processor core operates with 32 general purpose accumulators, allowing for a multitude of operations to be done. A system of multiplexers and a demultiplexer are used to route data between the memory, registers, and processing core. All of these components are controlled by the RISC core, where the signals are generated based on the machine code being inputted into the core.

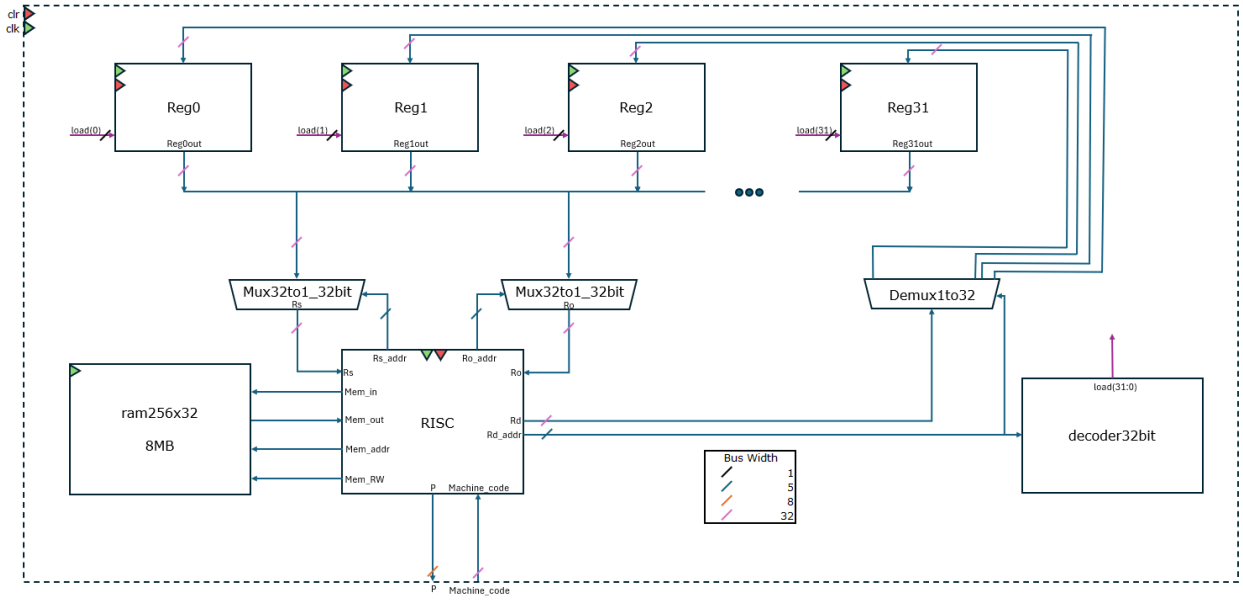


Figure 10: *RISC_Processor.vhd*

(Overview of the processor with the 32 accumulators, memory, multiplexers, and demultiplexer)

Finally, the RISC core contains the control unit, program counter, function unit (funit), multiplexers, and registers. The program counter increments the PROM, while also containing the ability to be loaded with a specific address from the PROM. The registers Rs and Ro load operands for arithmetic and boolean operations. The operands are loaded by a multiplexer, selecting from one of the 32 accumulators, an immediate value, or memory. The funit, where the operands are computed and the result is outputted. Finally, the output of the funit will lead to the output Rd, where it is then loaded into the desired accumulator. The main component that is controlling the entire RISC core is the RISC_CTRL. The control unit takes in machine code as an input, and outputs signals to other components depending on the machine code instruction. These signals include: the accumulator address loading into Ro, Rs, and Rd, the load signals for the Ro and Rs registers, immediate values extracted from the machine code, memory output with memory addresses, memory read/write control, and sending the opcode to the funit. **Figure 11**

represents a simulation of the RISC core running the command “LDUR X01,#089.” This command will load accumulator 1 with the immediate value of hexadecimal “89.”

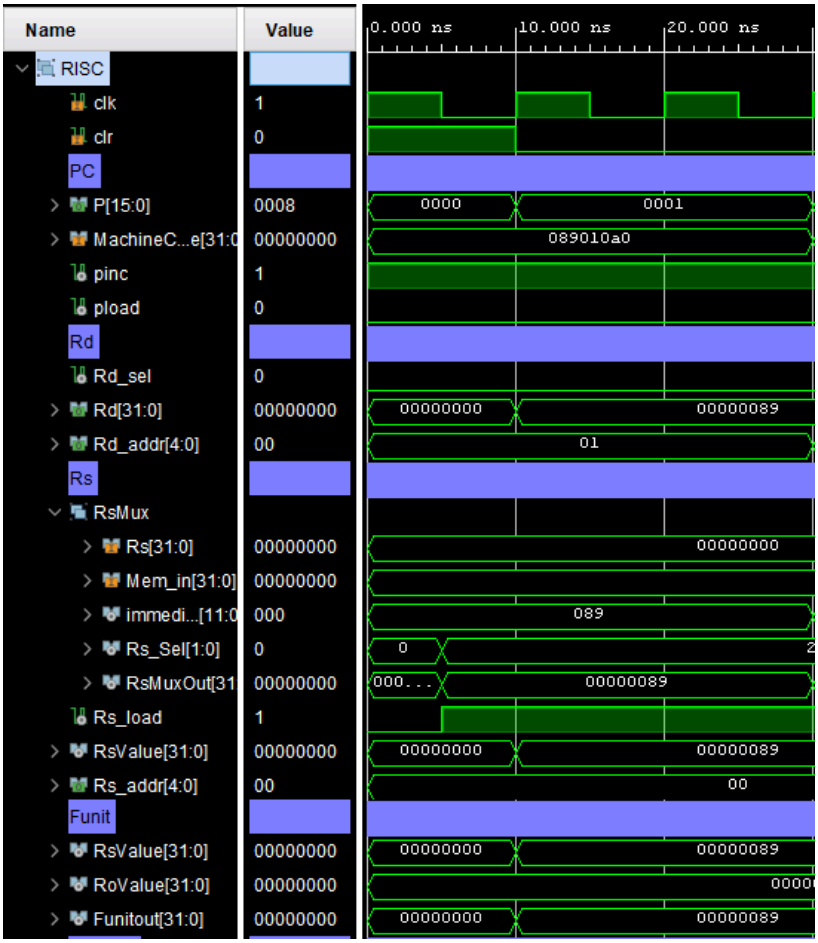


Figure 11: *RISC_Core Simulation*

(Simulation of LDUR X01,#089)

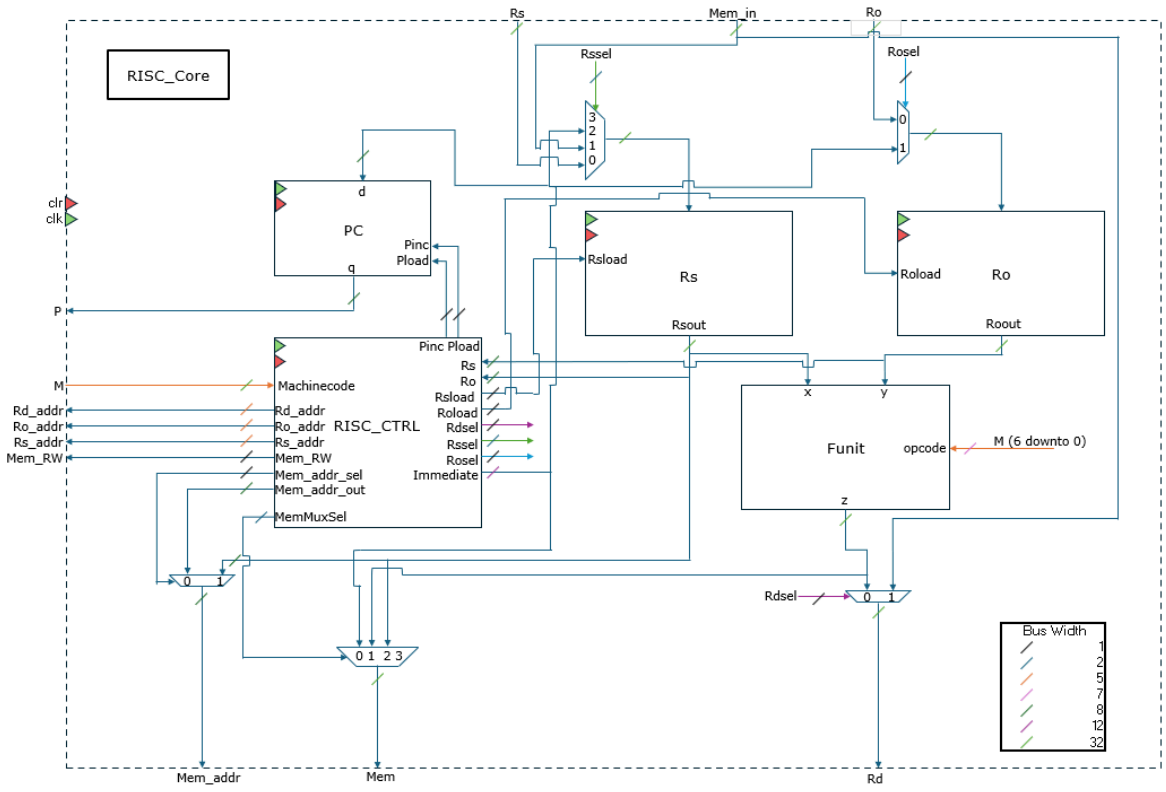


Figure 12: *RISC_Core.vhd*

(Overview of the RISC core)

5. Project Performance

The project is mostly functional, but some features that were envisioned from the start are missing. To begin, the project fully implements a functional text editor that takes inputs from a keyboard. The VGA monitor accurately displays keyboard inputs, increments the screen accordingly, and line breaks correctly. Additionally, the keyboard modules are capable of converting the PS/2 scan codes into ASCII pulses to be used by the VGA and PROM Programmer. Next, the PROM Programmer successfully takes the ASCII inputs, converts them into hexadecimal, and stores them into the PROM to be used by the RISC processor. Finally, the RISC processor processes the machine code as attended, and outputs the

expected values into the accumulators. The accumulator values are then capable of being viewed via the seven-segment display, switching between the accumulators with a 32 to 1 multiplexer.

Although the project functioned as intended, there are still several features that we were unable to implement due to lack of time. These features mainly pertained to the VGA display portion and the RISC Processor portion of the project.

Our initial goal was to improve the operation of the VGA display by adding more capabilities. The ability to execute the code and show the contents of all 32 registers on the screen by pressing the 'Esc' key was one feature we wanted to provide rather than switch 6. In order to make error correction easier, we also wanted to have a backspace option. Presently, the user must reset the processor and wipe the screen if they make a single mistake. The VGA display's ability to display both capital and lowercase characters was another feature we wanted to add. Only capital letters are available to the user at this time.

For the processing unit, we had three major improvements that we wanted to apply. The first major improvement was the implementation of an assembler. Having an assembler would allow us to write instructions in assembly language instead of directly inputting them as machine code. This would massively increase the usability of the project. The second improvement was aimed at the PROM. Currently, the lines of code written by the user are not erased after the processor is reset. This can lead to some problems if the user's initial program has more lines of code than their second one. To fix this we wanted to implement a function that would clear the PROM upon CPU reset, creating a blank canvas and ridding itself of the old code. The third improvement would be adding more RISC instruction variety. This includes branching and pseudo assembly code. These instructions would increase the usability of the project.

6. Conclusion

This project successfully demonstrates the implementation of a RISC-inspired processing system using the Nexys A7-100T FPGA board. By integrating a PS/2 keyboard decoder, VGA text generator, and a RISC processor, the system allows users to input, display, and execute custom machine code in real time. The development process involved overcoming significant challenges, such as ensuring timing synchronization between components and optimizing VGA rendering, which were resolved through iterative design and refinements.

Although the project achieved its primary objectives, there remains room for enhancement. Future improvements could include incorporating an assembler to simplify the programming process and expanding the VGA display's functionality to support more advanced graphical features. These refinements would make the system more user-friendly and versatile.

Overall, this project highlights the potential of FPGA-based designs for designing complex systems like processors. This led to a deeper understanding of digital systems and VHDL design, providing valuable insights into the intricacies of hardware-software integration.

7. References

1. Admiral Shark's Keyboards, "Keyboard Scancodes," Admiral Shark's Keyboards, https://sharktastica.co.uk/topics/scancodes#Set2. Accessed Nov. 2024
2. Chu, P. P. (2008). FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version, Chapter 13. Wiley-Blackwell.
3. Haskell, Richard, and Darrin Hanna. Advanced Digital Design Using Digilent FPGA Boards. LBE Books, LLC, 2016.
4. LLamocca, Daniel. "VHDL Coding for FPGAs." Dllamocca.org, dllamocca.org/VHDLforFPGAs.html. Accessed Nov. 2024.
5. Merrick, Russell. "Register Based FIFO in VHDL." NANDLAND, 9 June 2022, nandland.com/register-based-fifo/. Accessed Nov. 2024.
6. Waterman, Andrew, and Krste Asanović, editors. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2, Chapter 2. RISC-V Foundation, May 2017.

8. Appendix

1. Instruction Set

Yellow Highlighted means fully implemented

#	Instruction	Description	Opcode (7 bit)	Type
1	ADDD	Adds 2 regs	000_0000	R
2	SUBB	Subs 2 regs	000_0001	R
3	ADDI	Adds a reg with immediate	000_0010	I
4	SUBI	Subs a reg with immediate	000_0011	I
5	LUDR	Loads reg	010_0000	I
6	STUR	Stores reg	010_0001	S
7	LDRW	Loads reg w/ word	010_0010	I
8	STRW	Stores reg w/ word	010_0011	S
9	LDRH	Loads reg w/ half	010_0100	I
10	STRH	Stores reg w/ half	010_0101	S
11	LDRB	Loads reg w/ byte	010_0110	I
12	STRB	Stores reg w/ byte	010_0111	S
13	ALSR	Arithmetic shift right	000_0100	I
14	LSHL	Logical shift left	000_0101	I
15	LSHR	Logical shift right	000_0110	I
16	ANDD	AND 2 regs	000_0111	R
17	ANDI	AND a reg and immediate	000_1000	I
18	ORRR	OR 2 regs	000_1001	R
19	ORRI	OR a reg and immediate	000_1010	I
20	XORR	XOR 2 regs	000_1011	R
21	XORI	XOR a reg and immediate	000_1100	I
22	NOTR	Not a reg	000_1101	R
23	NOTI	Not an immediate	000_1110	I
24	TWOC	Takes 2's C of a reg	000_1111	I

25	MOVR	Transfer reg from to another	010_1000	I
26	INCR	Increment register	001_0000	I
27	DECR	Decrement register	001_0001	I
28	SGNX	Sign extends a signed word	001_0010	I
29	BRAI	Branch to another line w/ immediate	100_0000	B
30	BRAR	Branch to another line w/ register	100_0001	B
31	BEQZ	Branch if reg is equal to 0	100_0010	B
32	BNEZ	Branch if reg is not equal to 0	100_0100	B
33	BEQR	Branch if reg is equal to another reg	100_0101	B
34	BEQI	Branch if reg is equal to immediate	100_0110	B
35	BLTR	Branch if reg is less than register (unsigned)	100_0111	B
36	BLTI	Branch if reg is less than immediate (unsigned)	100_1000	B
37	BGTR	Branch if reg is greater than register (unsigned)	100_1001	B
38	BGTI	Branch if reg is greater than immediate (unsigned)	100_1010	B
39	NOPP	Skips clock cycle	001_0011	I
40	EXEC	Executes all programmed code	111_1111	
41	CLRR	Clears a register	001_0100	I
44	BSET	Sets specific bits in a register with an immediate	001_0111	I
45	BCLR	Clears specific bits in a register with an immediate	001_1000	I

		MAYBE		
	JSR	Jump to subroutine		
	RTS	Return from subroutine		
	SWAP	Swaps the value of 2 registers		

ADD X1, X2, X3 (X2 = 00001234) (X3 = 0000ABCD)

2. Assembly to Machine Code Conversion Sheet

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Misc							Ro					Rs					Misc			Rd					Opcode							
0	0	0	0	0	0	0	0	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	ADDD X1,X10,X11
0				0			B					5					0			0					8			0				
Immediate												Rs					Misc			Rd					Opcode							
1	0	1	0	1	0	0	1	1	0	1	1	0	0	1	1	1	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0	ADDI X2,X7,#A9B
A				9			B					3					9			1					0			2				
Address												Rs					Misc								Opcode							
0	0	0	1	0	0	1	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	STUR X04,@123
1				2			3					2					0			0					2			1				
Address												Rs					Misc			Rd					Opcode							
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	BRAI #01
0				0			1					0					1			0					4			0				

3. ASCII Table VHDL Package “ASCII_Table.vhd”

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

package ASCII_Table is
   subtype ascii_values is std_logic_vector (7 downto 0); -- 8-bit ASCII values in hex order

        -- Functions
        constant Carriage_Return : ascii_values := x"0D"; -- Carriage Return
        constant Backspace : ascii_values := x"08"; -- Backspace
        constant Space : ascii_values := x"20"; -- Space
        constant Escape : ascii_values := x"1B"; -- Escape

        -- Misc
        constant Hashtag_ASCII : ascii_values := x"23";
        constant Comma_ASCII : ascii_values := x"2C";
        constant Address_ASCII : ascii_values := x"40";

        -- Numbers
        constant Num0_ASCII : ascii_values := x"30"; -- '0'
        constant Num1_ASCII : ascii_values := x"31"; -- '1'
        constant Num2_ASCII : ascii_values := x"32"; -- '2'
        constant Num3_ASCII : ascii_values := x"33"; -- '3'
        constant Num4_ASCII : ascii_values := x"34"; -- '4'
        constant Num5_ASCII : ascii_values := x"35"; -- '5'
        constant Num6_ASCII : ascii_values := x"36"; -- '6'
        constant Num7_ASCII : ascii_values := x"37"; -- '7'
        constant Num8_ASCII : ascii_values := x"38"; -- '8'
        constant Num9_ASCII : ascii_values := x"39"; -- '9'

        -- Uppercase Letters
        constant UpperA_ASCII : ascii_values := x"41"; -- 'A'
        constant UpperB_ASCII : ascii_values := x"42"; -- 'B'
        constant UpperC_ASCII : ascii_values := x"43"; -- 'C'
        constant UpperD_ASCII : ascii_values := x"44"; -- 'D'
        constant UpperE_ASCII : ascii_values := x"45"; -- 'E'
        constant UpperF_ASCII : ascii_values := x"46"; -- 'F'
        constant UpperG_ASCII : ascii_values := x"47"; -- 'G'
        constant UpperH_ASCII : ascii_values := x"48"; -- 'H'
        constant UpperI_ASCII : ascii_values := x"49"; -- 'I'
```

```
        constant UpperJ_ASCII : ascii_values := x"4A"; -- 'J'
        constant UpperK_ASCII : ascii_values := x"4B"; -- 'K'
        constant UpperL_ASCII : ascii_values := x"4C"; -- 'L'
        constant UpperM_ASCII : ascii_values := x"4D"; -- 'M'
        constant UpperN_ASCII : ascii_values := x"4E"; -- 'N'
        constant UpperO_ASCII : ascii_values := x"4F"; -- 'O'
        constant UpperP_ASCII : ascii_values := x"50"; -- 'P'
        constant UpperQ_ASCII : ascii_values := x"51"; -- 'Q'
        constant UpperR_ASCII : ascii_values := x"52"; -- 'R'
        constant UpperS_ASCII : ascii_values := x"53"; -- 'S'
        constant UpperT_ASCII : ascii_values := x"54"; -- 'T'
        constant UpperU_ASCII : ascii_values := x"55"; -- 'U'
        constant UpperV_ASCII : ascii_values := x"56"; -- 'V'
        constant UpperW_ASCII : ascii_values := x"57"; -- 'W'
        constant UpperX_ASCII : ascii_values := x"58"; -- 'X'
        constant UpperY_ASCII : ascii_values := x"59"; -- 'Y'
        constant UpperZ_ASCII : ascii_values := x"5A"; -- 'Z'

        -- Lowercase Letters
        constant LowerA_ASCII : ascii_values := x"61"; -- 'a'
        constant LowerB_ASCII : ascii_values := x"62"; -- 'b'
        constant LowerC_ASCII : ascii_values := x"63"; -- 'c'
        constant LowerD_ASCII : ascii_values := x"64"; -- 'd'
        constant LowerE_ASCII : ascii_values := x"65"; -- 'e'
        constant LowerF_ASCII : ascii_values := x"66"; -- 'f'
        constant LowerG_ASCII : ascii_values := x"67"; -- 'g'
        constant LowerH_ASCII : ascii_values := x"68"; -- 'h'
        constant LowerI_ASCII : ascii_values := x"69"; -- 'i'
        constant LowerJ_ASCII : ascii_values := x"6A"; -- 'j'
        constant LowerK_ASCII : ascii_values := x"6B"; -- 'k'
        constant LowerL_ASCII : ascii_values := x"6C"; -- 'l'
        constant LowerM_ASCII : ascii_values := x"6D"; -- 'm'
        constant LowerN_ASCII : ascii_values := x"6E"; -- 'n'
        constant LowerO_ASCII : ascii_values := x"6F"; -- 'o'
        constant LowerP_ASCII : ascii_values := x"70"; -- 'p'
        constant LowerQ_ASCII : ascii_values := x"71"; -- 'q'
        constant LowerR_ASCII : ascii_values := x"72"; -- 'r'
        constant LowerS_ASCII : ascii_values := x"73"; -- 's'
        constant LowerT_ASCII : ascii_values := x"74"; -- 't'
        constant LowerU_ASCII : ascii_values := x"75"; -- 'u'
        constant LowerV_ASCII : ascii_values := x"76"; -- 'v'
        constant LowerW_ASCII : ascii_values := x"77"; -- 'w'
        constant LowerX_ASCII : ascii_values := x"78"; -- 'x'
        constant LowerY_ASCII : ascii_values := x"79"; -- 'y'
        constant LowerZ_ASCII : ascii_values := x"7A"; -- 'z'

end ASCII_Table;
```

4. RISC Opcode Instruction Set VHDL Package “RISC_Instruction_Set.vhd”

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

package RISC_Instruction_Set is
   subtype opcodes is std_logic_vector(6 downto 0);

   -- Arithmetic Instructions
   constant ADDD  : opcodes := "0000000";  -- Adds 2 registers
   constant SUBB  : opcodes := "0000001";  -- Subtracts 2 registers
   constant ADDI  : opcodes := "0000010";  -- Adds register and immediate
   constant SUBI  : opcodes := "0000011";  -- Subtracts register and immediate

   -- Load and Store Instructions
   constant LUDR  : opcodes := "0100000";  -- Load register
   constant STUR  : opcodes := "0100001";  -- Store register
   constant LDRW  : opcodes := "0100010";  -- Load word
   constant STRW  : opcodes := "0100011";  -- Store word
   constant LDRH  : opcodes := "0100100";  -- Load half-word
   constant STRH  : opcodes := "0100101";  -- Store half-word
   constant LDRB  : opcodes := "0100110";  -- Load byte
   constant STRB  : opcodes := "0100111";  -- Store byte

   -- Logical and Shift Instructions
   constant ANDD  : opcodes := "0000111";  -- AND two registers
   constant ANDI  : opcodes := "0001000";  -- AND register and immediate
   constant ORRR  : opcodes := "0001001";  -- OR two registers
   constant ORRI  : opcodes := "0001010";  -- OR register and immediate
   constant XORR  : opcodes := "0001011";  -- XOR two registers
   constant XORI  : opcodes := "0001100";  -- XOR register and immediate
   constant NOTR  : opcodes := "0001101";  -- NOT a register
   constant NOTI  : opcodes := "0001110";  -- NOT an immediate
   constant ALSR  : opcodes := "0000100";  -- Arithmetic shift right
   constant LSHL  : opcodes := "0000101";  -- Logical shift left
   constant LSHR  : opcodes := "0000110";  -- Logical shift right
   constant TWOC  : opcodes := "0001111";  -- Two's complement

   -- Increment, Decrement, and Sign Extend
   constant INCR  : opcodes := "0010000";  -- Increment register
   constant DECR  : opcodes := "0010001";  -- Decrement register
```

```
    constant SGNX  : opcodes := "0010010";  -- Sign extend
    constant MOVR  : opcodes := "0101000";  -- Move register

    -- Branch Instructions
    constant BRAI  : opcodes := "1000000";  -- Branch with immediate
    constant BRAR  : opcodes := "1000001";  -- Branch with register
    constant BEQZ  : opcodes := "1000010";  -- Branch if equal to zero
    constant BNEZ  : opcodes := "1000100";  -- Branch if not equal to zero
    constant BEQR  : opcodes := "1000101";  -- Branch if registers are equal
    constant BEQI  : opcodes := "1000110";  -- Branch if equal to immediate
    constant BLTR  : opcodes := "1000111";  -- Branch if less than register
    constant BLTI  : opcodes := "1001000";  -- Branch if less than immediate
    constant BGTR  : opcodes := "1001001";  -- Branch if greater than register
    constant BGTI  : opcodes := "1001010";  -- Branch if greater than immediate

    -- Other Instructions
    constant NOPP  : opcodes := "0010011";  -- No operation
    constant EXEC  : opcodes := "1111111";  -- Execute all code
    constant CLRR  : opcodes := "0010100";  -- Clear register
    constant BSET  : opcodes := "0010111";  -- Set specific bits
    constant BCLR  : opcodes := "0011000";  -- Clear specific bits

    -- Placeholder for future instructions
    constant JSR   : opcodes := "1110000";  -- Jump to subroutine
    constant RTS   : opcodes := "1110001";  -- Return from subroutine
    constant SWAP  : opcodes := "1110010";  -- Swap registers

end RISC_Instruction_Set;
```

5. RISC Core “RISC_Core.vhd”

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity RISC is
    Port (
          -- Clk
          clk : in std_logic;
          clr : in std_logic;
          -- PC
          P : out std_logic_vector (15 downto 0);
          -- Registers
          Rs : in STD_LOGIC_VECTOR (31 downto 0);
          Rs_addr : out STD_LOGIC_VECTOR (4 downto 0);
          Ro : in STD_LOGIC_VECTOR (31 downto 0);
          Ro_addr : out STD_LOGIC_VECTOR (4 downto 0);
          Rd : out STD_LOGIC_VECTOR (31 downto 0);
          Rd_addr : out STD_LOGIC_VECTOR (4 downto 0);
          -- ROM
          MachineCode : in STD_LOGIC_VECTOR (31 downto 0);
          -- Memory
          Mem_in : in std_logic_vector (31 downto 0);
          Mem_out : out STD_LOGIC_VECTOR (31 downto 0);
          Mem_addr : out STD_LOGIC_VECTOR (7 downto 0);
          Mem_RW : out STD_LOGIC
          );
end RISC;

architecture Behavioral of RISC is

component nbitregister
  generic(N:integer);
  Port (load : in std_logic;
        clk : in std_logic;
        clr : in std_logic;
        d : in std_logic_vector (N-1 downto 0);
        q : out std_logic_vector (N-1 downto 0));
end component;

component RISC_ctrl
```

```
  Port (
        -- Clk
        clk : in std_logic;
        clr : in std_logic;
        -- Machine Code
        MachineCode : in std_logic_vector (31 downto 0);
        -- Memory
        Mem_addr_sel : out std_logic;
        Mem_addr : out std_logic_vector (7 downto 0);
        Mem_Mux_sel : out std_logic_vector (1 downto 0);
--        Mem_load : out std_logic;
--        Mem_addr_load : out std_logic;
        Mem_RW : out std_logic;
        -- Rs
        Rs : in std_logic_vector (31 downto 0);
        Rs_load : out std_logic;
        Rs_sel : out std_logic_vector (1 downto 0);
        Rs_addr : out std_logic_vector (4 downto 0);
--        Rs_addr_load : in std_logic;
        -- Ro
        Ro : in std_logic_vector (31 downto 0);
        Ro_load : out std_logic;
        Ro_sel : out std_logic;
        Immediate_Value : out std_logic_vector (11 downto 0);
        Ro_addr : out std_logic_vector (4 downto 0);
--        Ro_addr_load : in std_logic;
        -- Rd
--        Rd_load : out std_logic;
        Rd_addr : out std_logic_vector (4 downto 0);
        Rd_sel : out std_logic;
--        Rd_addr_load : in std_logic;
        -- PC
        pinc : out std_logic;
        pload : out std_logic;
        -- MISC
        error : out std_logic
        );
end component;

component Funit
    Port ( a : in STD_LOGIC_VECTOR (31 downto 0);
           b : in STD_LOGIC_VECTOR (31 downto 0);
           opcode : in STD_LOGIC_VECTOR (6 downto 0);
           y : out STD_LOGIC_VECTOR (31 downto 0);
           error : out std_logic
            );
end component;

component mux2to1_nbit
  generic(N:integer);
```

```
  Port (
       a : in std_logic_vector (N - 1 downto 0);
       b : in std_logic_vector (N - 1 downto 0);
       sel : in std_logic;
       x : out std_logic_vector (N - 1 downto 0)
       );
end component;

component mux4to1_32bit
  Port (
       a : in std_logic_vector (31 downto 0);
       b : in std_logic_vector (31 downto 0);
       c : in std_logic_vector (31 downto 0);
       d : in std_logic_vector (31 downto 0);
       sel : in std_logic_vector (1 downto 0);
       x : out std_logic_vector (31 downto 0)
       );
end component;

component PC
  Port (
       clk : in std_logic;
       clr : in std_logic;
       d : in std_logic_vector (15 downto 0);
       inc : in std_logic;
       load : in std_logic;
       q : out std_logic_vector (15 downto 0)
  );
end component;

-- Signals
signal immediateValue : std_logic_vector(11 downto 0);
signal Rs_Sel, Mem_mux_sel : std_logic_vector (1 downto 0);
signal Rs_load, Ro_sel, Ro_load, pinc, pload, Mem_addr_sel, Rd_sel, clk50M : std_logic;
signal RsMuxOut, RsValue, RoMuxOut, RoValue, Funitout : std_logic_vector (31 downto 0);
signal Mem_addr_sig : std_logic_vector (7 downto 0);

-- Errors
signal FunitError, CTRLError : std_logic;

begin

RsMUX : mux4to1_32bit port map  (a => Rs,
                         b => Mem_in,
                         c => x"00000" & immediateValue,
                         d => x"00000000",
                         sel => Rs_Sel,
                         x => RsMuxOut);

RsReg : nbitregister generic map    (N => 32)
```

```
                port map    (clk => clk,
                            clr => clr,
                            load => Rs_load,
                            d => RsMuxOut,
                            q => RsValue);

RoMUX : mux2to1_nbit generic map    (N => 32)
                port map    (a => Ro,
                            b => x"00000" & immediateValue,
                            sel => Ro_sel,
                            x => RoMuxOut);

RoReg : nbitregister generic map    (N => 32)
                port map    (clk => clk,
                            clr => clr,
                            load => Ro_load,
                            d => RoMuxOut,
                            q => RoValue);

ProgramCounter : PC port map    (clk => clk,
                        clr => clr,
                        d => x"0" & immediateValue,
                        inc => pinc,
                        load => pload,
                        q => P);

Funit1 : Funit port map (a => RsValue,
                b => RoValue,
                opcode => MachineCode (6 downto 0),
                y => Funitout,
                error => FunitError);

CTRL : RISC_ctrl port map   (clk => clk,
                    clr => clr,
                    MachineCode => MachineCode,
                    Mem_addr_sel => Mem_addr_sel,
                    Mem_addr => Mem_addr_sig,
                    Mem_mux_sel => Mem_mux_sel,
                    Mem_RW => Mem_RW,
                    Rs_load => Rs_load,
                    Rs_sel => Rs_sel,
                    Ro_load => Ro_load,
                    Ro_sel => Ro_sel,
                    Immediate_Value => immediateValue,
                    pinc => pinc,
                    pload => pload,
                    error => CTRLError,
                    Rs_addr => Rs_addr,
                    Ro_addr => Ro_addr,
                    Rd_addr => Rd_addr,
```

```
                    Rs => RsValue,
                    Ro => RoValue,
                    Rd_sel => Rd_sel
                    );

MemAddrMux : mux2to1_nbit generic map   (N => 8)
                    port map    (a => Mem_addr_sig,
                                b => RsValue(7 downto 0),
                                sel => Mem_addr_sel,
                                x => Mem_addr);

MemMux : mux4to1_32bit port map (a => x"00000" & immediateValue,
                        b => Funitout,
                        c => RsValue,
                        d => x"00000000",
                        sel => Mem_mux_sel,
                        x => Mem_out);

RdMux : mux2to1_nbit generic map    (N => 32)
                port map    (a => Funitout,
                            b => Mem_in,
                            sel => Rd_sel,
                            x => Rd);

end Behavioral;
```

6. RISC Processor w/ accumulators “RISC_Processor.vhd”

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity RISC_Processor is
  Port (
        clk : in std_logic;
        clr : in std_logic;
        MachineCode : in std_logic_vector (31 downto 0);
        P : out std_logic_vector (15 downto 0);
        X0_out : out std_logic_vector (31 downto 0);
        X1_out : out std_logic_vector (31 downto 0);
        X2_out : out std_logic_vector (31 downto 0);
        X3_out : out std_logic_vector (31 downto 0);
        X4_out : out std_logic_vector (31 downto 0);
        X5_out : out std_logic_vector (31 downto 0);
        X6_out : out std_logic_vector (31 downto 0);
        X7_out : out std_logic_vector (31 downto 0);
        X8_out : out std_logic_vector (31 downto 0);
        X9_out : out std_logic_vector (31 downto 0);
        X10_out : out std_logic_vector (31 downto 0);
        X11_out : out std_logic_vector (31 downto 0);
        X12_out : out std_logic_vector (31 downto 0);
        X13_out : out std_logic_vector (31 downto 0);
        X14_out : out std_logic_vector (31 downto 0);
        X15_out : out std_logic_vector (31 downto 0);
        X16_out : out std_logic_vector (31 downto 0);
        X17_out : out std_logic_vector (31 downto 0);
        X18_out : out std_logic_vector (31 downto 0);
        X19_out : out std_logic_vector (31 downto 0);
        X20_out : out std_logic_vector (31 downto 0);
        X21_out : out std_logic_vector (31 downto 0);
        X22_out : out std_logic_vector (31 downto 0);
        X23_out : out std_logic_vector (31 downto 0);
        X24_out : out std_logic_vector (31 downto 0);
        X25_out : out std_logic_vector (31 downto 0);
        X26_out : out std_logic_vector (31 downto 0);
        X27_out : out std_logic_vector (31 downto 0);
        X28_out : out std_logic_vector (31 downto 0);
        X29_out : out std_logic_vector (31 downto 0);
```

```
        X30_out : out std_logic_vector (31 downto 0);
        X31_out : out std_logic_vector (31 downto 0)
        );
end RISC_Processor;


architecture Behavioral of RISC_Processor is

component nbitregister
  generic(N:integer);
  Port (load : in std_logic;
        clk : in std_logic;
        clr : in std_logic;
        d : in std_logic_vector (N-1 downto 0);
        q : out std_logic_vector (N-1 downto 0));
end component;

component mux32to1_32bit
  Port (
        a : in std_logic_vector (31 downto 0); -- 1
        b : in std_logic_vector (31 downto 0); -- 2
        c : in std_logic_vector (31 downto 0); -- 3
        d : in std_logic_vector (31 downto 0); -- 4
        e : in std_logic_vector (31 downto 0); -- 5
        f : in std_logic_vector (31 downto 0); -- 6
        g : in std_logic_vector (31 downto 0); -- 7
        h : in std_logic_vector (31 downto 0); -- 8
        i : in std_logic_vector (31 downto 0); -- 9
        j : in std_logic_vector (31 downto 0); -- 10
        k : in std_logic_vector (31 downto 0); -- 11
        l : in std_logic_vector (31 downto 0); -- 12
        m : in std_logic_vector (31 downto 0); -- 13
        n : in std_logic_vector (31 downto 0); -- 14
        o : in std_logic_vector (31 downto 0); -- 15
        p : in std_logic_vector (31 downto 0); -- 16
        q : in std_logic_vector (31 downto 0); -- 17
        r : in std_logic_vector (31 downto 0); -- 18
        s : in std_logic_vector (31 downto 0); -- 19
        t : in std_logic_vector (31 downto 0); -- 20
        u : in std_logic_vector (31 downto 0); -- 21
        v : in std_logic_vector (31 downto 0); -- 22
        w : in std_logic_vector (31 downto 0); -- 23
        x : in std_logic_vector (31 downto 0); -- 24
        y : in std_logic_vector (31 downto 0); -- 25
        z : in std_logic_vector (31 downto 0); -- 26
        aa : in std_logic_vector (31 downto 0); -- 27
        bb : in std_logic_vector (31 downto 0); -- 28
        cc : in std_logic_vector (31 downto 0); -- 29
        dd : in std_logic_vector (31 downto 0); -- 30
        ee : in std_logic_vector (31 downto 0); -- 31
```

```
        ff : in std_logic_vector (31 downto 0); -- 32
        sel : in std_logic_vector (4 downto 0);
        zout : out std_logic_vector (31 downto 0)
        );
end component;

component demux1to32_32bit
  Port (
        zin : in std_logic_vector (31 downto 0);
        sel : in std_logic_vector (4 downto 0);
        a : out std_logic_vector (31 downto 0); -- 1
        b : out std_logic_vector (31 downto 0); -- 2
        c : out std_logic_vector (31 downto 0); -- 3
        d : out std_logic_vector (31 downto 0); -- 4
        e : out std_logic_vector (31 downto 0); -- 5
        f : out std_logic_vector (31 downto 0); -- 6
        g : out std_logic_vector (31 downto 0); -- 7
        h : out std_logic_vector (31 downto 0); -- 8
        i : out std_logic_vector (31 downto 0); -- 9
        j : out std_logic_vector (31 downto 0); -- 10
        k : out std_logic_vector (31 downto 0); -- 11
        l : out std_logic_vector (31 downto 0); -- 12
        m : out std_logic_vector (31 downto 0); -- 13
        n : out std_logic_vector (31 downto 0); -- 14
        o : out std_logic_vector (31 downto 0); -- 15
        p : out std_logic_vector (31 downto 0); -- 16
        q : out std_logic_vector (31 downto 0); -- 17
        r : out std_logic_vector (31 downto 0); -- 18
        s : out std_logic_vector (31 downto 0); -- 19
        t : out std_logic_vector (31 downto 0); -- 20
        u : out std_logic_vector (31 downto 0); -- 21
        v : out std_logic_vector (31 downto 0); -- 22
        w : out std_logic_vector (31 downto 0); -- 23
        x : out std_logic_vector (31 downto 0); -- 24
        y : out std_logic_vector (31 downto 0); -- 25
        z : out std_logic_vector (31 downto 0); -- 26
        aa : out std_logic_vector (31 downto 0); -- 27
        bb : out std_logic_vector (31 downto 0); -- 28
        cc : out std_logic_vector (31 downto 0); -- 29
        dd : out std_logic_vector (31 downto 0); -- 30
        ee : out std_logic_vector (31 downto 0); -- 31
        ff : out std_logic_vector (31 downto 0) -- 32
        );
end component;

component ram256x32
  PORT (
    clka : IN STD_LOGIC;
    wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
```

```
   dina : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
   douta : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
  );
END component;

component RISC
   Port (
         -- Clk
         clk : in std_logic;
         clr : in std_logic;
         -- PC
         P : out std_logic_vector (15 downto 0);
         -- Registers
         Rs : in STD_LOGIC_VECTOR (31 downto 0);
         Rs_addr : out STD_LOGIC_VECTOR (4 downto 0);
         Ro : in STD_LOGIC_VECTOR (31 downto 0);
         Ro_addr : out STD_LOGIC_VECTOR (4 downto 0);
         Rd : out STD_LOGIC_VECTOR (31 downto 0);
         Rd_addr : out STD_LOGIC_VECTOR (4 downto 0);
         -- ROM
         MachineCode : in STD_LOGIC_VECTOR (31 downto 0);
         -- Memory
         Mem_in : in std_logic_vector (31 downto 0);
         Mem_out : out STD_LOGIC_VECTOR (31 downto 0);
         Mem_addr : out STD_LOGIC_VECTOR (7 downto 0);
         Mem_RW : out STD_LOGIC
         );
end component;

component decoder_32bit
  Port (
       a : in std_logic_vector (4 downto 0);
       x : out std_logic_vector (31 downto 0)
       );
end component;

type my_array is array (natural range<>) of std_logic_vector (31 downto 0);
signal din, qout : my_array (0 to 31);
signal acc_load : std_logic_vector (0 to 31);
signal Rs_addr, Ro_addr, Rd_addr : std_logic_vector (4 downto 0);
signal RsMuxOut, RoMuxOut, RdOut, ramin, ramout : std_logic_vector (31 downto 0);
signal Mem_RW : std_logic;
signal wea : std_logic_vector (0 downto 0);
signal Mem_addr : std_logic_vector (7 downto 0);
signal acc_load_combined, next_acc_load_combined : std_logic_vector (31 downto 0);

begin

zsr : nbitregister generic map  (N => 32)
                port map    (clk => clk,
```

```
                        clr => clr,
                        load => '0',
                        d => x"00000000",
                        q => qout(0));

accumulatorloop : for i in 1 to 31 generate
   reg : nbitregister generic map  (N => 32)
                port map    (clk => clk,
                        clr => clr,
                        load => acc_load(i),
                        d => din(i),
                        q => qout(i));
end generate;

RsMux : mux32to1_32bit port map (a => qout(0),
                        b => qout(1),
                        c => qout(2),
                        d => qout(3),
                        e => qout(4),
                        f => qout(5),
                        g => qout(6),
                        h => qout(7),
                        i => qout(8),
                        j => qout(9),
                        k => qout(10),
                        l => qout(11),
                        m => qout(12),
                        n => qout(13),
                        o => qout(14),
                        p => qout(15),
                        q => qout(16),
                        r => qout(17),
                        s => qout(18),
                        t => qout(19),
                        u => qout(20),
                        v => qout(21),
                        w => qout(22),
                        x => qout(23),
                        y => qout(24),
                        z => qout(25),
                        aa => qout(26),
                        bb => qout(27),
                        cc => qout(28),
                        dd => qout(29),
                        ee => qout(30),
                        ff => qout(31),
                        sel => Rs_addr,
                        zout => RsMuxOut);

RoMux : mux32to1_32bit port map (a => qout(0),
```

```
                        b => qout(1),
                        c => qout(2),
                        d => qout(3),
                        e => qout(4),
                        f => qout(5),
                        g => qout(6),
                        h => qout(7),
                        i => qout(8),
                        j => qout(9),
                        k => qout(10),
                        l => qout(11),
                        m => qout(12),
                        n => qout(13),
                        o => qout(14),
                        p => qout(15),
                        q => qout(16),
                        r => qout(17),
                        s => qout(18),
                        t => qout(19),
                        u => qout(20),
                        v => qout(21),
                        w => qout(22),
                        x => qout(23),
                        y => qout(24),
                        z => qout(25),
                        aa => qout(26),
                        bb => qout(27),
                        cc => qout(28),
                        dd => qout(29),
                        ee => qout(30),
                        ff => qout(31),
                        sel => Ro_addr,
                        zout => RoMuxOut);

RdDemux : demux1to32_32bit port map (zin => RdOut,
                            sel => Rd_addr,
                            a => din(0),
                            b => din(1),
                            c => din(2),
                            d => din(3),
                            e => din(4),
                            f => din(5),
                            g => din(6),
                            h => din(7),
                            i => din(8),
                            j => din(9),
                            k => din(10),
                            l => din(11),
                            m => din(12),
                            n => din(13),
```

```
                        o => din(14),
                        p => din(15),
                        q => din(16),
                        r => din(17),
                        s => din(18),
                        t => din(19),
                        u => din(20),
                        v => din(21),
                        w => din(22),
                        x => din(23),
                        y => din(24),
                        z => din(25),
                        aa => din(26),
                        bb => din(27),
                        cc => din(28),
                        dd => din(29),
                        ee => din(30),
                        ff => din(31));

decoder : decoder_32bit port map    (a => rd_addr,
                        x => acc_load_combined);

accloop : for i in 0 to 31 generate
   acc_load(i) <= acc_load_combined(i);
end generate;


Processor : RISC port map   (clk => clk,
                    clr => clr,
                    P => P,
                    Rs => RsMuxOut,
                    Rs_addr => Rs_addr,
                    Ro => RoMuxOut,
                    Ro_addr => Ro_addr,
                    Rd => RdOut,
                    Rd_addr => Rd_addr,
                    MachineCode => MachineCode,
                    Mem_in => ramin,
                    Mem_out => ramout,
                    Mem_addr => Mem_addr,
                    Mem_RW => Mem_Rw);

wea(0) <= Mem_RW;

Memory : ram256x32 port map (clka => clk,
                    wea => wea,
                    addra => Mem_addr,
                    dina => ramout,
                    douta => ramin);
```

```
X0_out <= qout(0);
X1_out <= qout(1);
X2_out <= qout(2);
X3_out <= qout(3);
X4_out <= qout(4);
X5_out <= qout(5);
X6_out <= qout(6);
X7_out <= qout(7);
X8_out <= qout(8);
X9_out <= qout(9);
X10_out <= qout(10);
X11_out <= qout(11);
X12_out <= qout(12);
X13_out <= qout(13);
X14_out <= qout(14);
X15_out <= qout(15);
X16_out <= qout(16);
X17_out <= qout(17);
X18_out <= qout(18);
X19_out <= qout(19);
X20_out <= qout(20);
X21_out <= qout(21);
X22_out <= qout(22);
X23_out <= qout(23);
X24_out <= qout(24);
X25_out <= qout(25);
X26_out <= qout(26);
X27_out <= qout(27);
X28_out <= qout(28);
X29_out <= qout(29);
X30_out <= qout(30);
X31_out <= qout(31);

end Behavioral;
```

7. RISC Final Assembly “RISC_Final_Assembly.vhd”

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.all;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity RISC_Final_Assembly is
  Port (
        clk : in std_logic;
        clr : in std_logic;
        ascii : in std_logic_vector (7 downto 0);
        processor_load : in std_logic;
        X0_out : out std_logic_vector (31 downto 0);
        X1_out : out std_logic_vector (31 downto 0);
        X2_out : out std_logic_vector (31 downto 0);
        X3_out : out std_logic_vector (31 downto 0);
        X4_out : out std_logic_vector (31 downto 0);
        X5_out : out std_logic_vector (31 downto 0);
        X6_out : out std_logic_vector (31 downto 0);
        X7_out : out std_logic_vector (31 downto 0);
        X8_out : out std_logic_vector (31 downto 0);
        X9_out : out std_logic_vector (31 downto 0);
        X10_out : out std_logic_vector (31 downto 0);
        X11_out : out std_logic_vector (31 downto 0);
        X12_out : out std_logic_vector (31 downto 0);
        X13_out : out std_logic_vector (31 downto 0);
        X14_out : out std_logic_vector (31 downto 0);
        X15_out : out std_logic_vector (31 downto 0);
        X16_out : out std_logic_vector (31 downto 0);
        X17_out : out std_logic_vector (31 downto 0);
        X18_out : out std_logic_vector (31 downto 0);
        X19_out : out std_logic_vector (31 downto 0);
        X20_out : out std_logic_vector (31 downto 0);
        X21_out : out std_logic_vector (31 downto 0);
        X22_out : out std_logic_vector (31 downto 0);
        X23_out : out std_logic_vector (31 downto 0);
        X24_out : out std_logic_vector (31 downto 0);
        X25_out : out std_logic_vector (31 downto 0);
        X26_out : out std_logic_vector (31 downto 0);
        X27_out : out std_logic_vector (31 downto 0);
        X28_out : out std_logic_vector (31 downto 0);
```

```
        X29_out : out std_logic_vector (31 downto 0);
        X30_out : out std_logic_vector (31 downto 0);
        X31_out : out std_logic_vector (31 downto 0)
         );
end RISC_Final_Assembly;

architecture Behavioral of RISC_Final_Assembly is

component RISC_Processor
  Port (
        clk : in std_logic;
        clr : in std_logic;
        MachineCode : in std_logic_vector (31 downto 0);
        P : out std_logic_vector (15 downto 0);
        X0_out : out std_logic_vector (31 downto 0);
        X1_out : out std_logic_vector (31 downto 0);
        X2_out : out std_logic_vector (31 downto 0);
        X3_out : out std_logic_vector (31 downto 0);
        X4_out : out std_logic_vector (31 downto 0);
        X5_out : out std_logic_vector (31 downto 0);
        X6_out : out std_logic_vector (31 downto 0);
        X7_out : out std_logic_vector (31 downto 0);
        X8_out : out std_logic_vector (31 downto 0);
        X9_out : out std_logic_vector (31 downto 0);
        X10_out : out std_logic_vector (31 downto 0);
        X11_out : out std_logic_vector (31 downto 0);
        X12_out : out std_logic_vector (31 downto 0);
        X13_out : out std_logic_vector (31 downto 0);
        X14_out : out std_logic_vector (31 downto 0);
        X15_out : out std_logic_vector (31 downto 0);
        X16_out : out std_logic_vector (31 downto 0);
        X17_out : out std_logic_vector (31 downto 0);
        X18_out : out std_logic_vector (31 downto 0);
        X19_out : out std_logic_vector (31 downto 0);
        X20_out : out std_logic_vector (31 downto 0);
        X21_out : out std_logic_vector (31 downto 0);
        X22_out : out std_logic_vector (31 downto 0);
        X23_out : out std_logic_vector (31 downto 0);
        X24_out : out std_logic_vector (31 downto 0);
        X25_out : out std_logic_vector (31 downto 0);
        X26_out : out std_logic_vector (31 downto 0);
        X27_out : out std_logic_vector (31 downto 0);
        X28_out : out std_logic_vector (31 downto 0);
        X29_out : out std_logic_vector (31 downto 0);
        X30_out : out std_logic_vector (31 downto 0);
        X31_out : out std_logic_vector (31 downto 0)
        );
end component;

component PROM_Programmer
```

```
    Port (
        clk : in STD_LOGIC;
        clr : in STD_LOGIC;
        ascii_in : in STD_LOGIC_VECTOR(7 downto 0);
        concadinated : out STD_LOGIC_VECTOR(63 downto 0);
        done : out STD_LOGIC -- Goes high when concatenation is complete
    );
end component;

component concadinated_to_hex
    Port (
        clk : in std_logic;
        concadinated : in STD_LOGIC_VECTOR(63 downto 0);
        hex : out STD_LOGIC_VECTOR(31 downto 0)
    );
end component;

component PROM64x32
  PORT (
    clka : IN STD_LOGIC;
    wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    addra : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
    dina : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
  );
END component;

component Tristate_Buffer
    Port ( d : in STD_LOGIC;
           en : in STD_LOGIC;
           q : out STD_LOGIC);
end component;

component mux2to1_nbit
  generic(N:integer);
  Port (
        a : in std_logic_vector (N - 1 downto 0);
        b : in std_logic_vector (N - 1 downto 0);
        sel : in std_logic;
        x : out std_logic_vector (N - 1 downto 0)
        );
end component;

-- Accumulator array
type my_array is array (natural range<>) of std_logic_vector (31 downto 0);
signal qout : my_array (0 to 31);

-- Signals
signal enqueue, enter, assembler_done, clk_register : std_logic;
signal wea : std_logic_vector (0 downto 0);
```

```
signal prom_count, addr_mux_out : std_logic_vector (5 downto 0);
signal ascii_sig : std_logic_vector (7 downto 0);
signal P : std_logic_vector (15 downto 0);
signal machine_code, processor_stream : std_logic_vector (31 downto 0);
signal concadinated : std_logic_vector (63 downto 0);

begin

PROM_counter : process(clk, clr)
begin

   if clr = '1' then
      prom_count <= "000001";
   elsif clk = '1' and clk'event then
      if assembler_done = '1' then
         prom_count <= prom_count + 1;
      end if;
   end if;

end process;

Assembler1 : PROM_Programmer port map   (clk => clk,
                                clr => clr,
                                ascii_in => ascii,
                                concadinated => concadinated,
                                done => assembler_done
                                );

Assembler2 : concadinated_to_hex port map   (clk => clk,
                                   concadinated => concadinated,
                                   hex => machine_code);

addr_mux : mux2to1_nbit generic map (N => 6)
                  port map    (a => prom_count,
                               b => P (5 downto 0),
                               sel => processor_load,
                               x => addr_mux_out);

wea(0) <= assembler_done;

PROM : PROM64x32 port map   (clka => clk,
                        wea => wea,
                        addra => addr_mux_out,
                        dina => machine_code,
                        douta => processor_stream);

clkbuffer : Tristate_Buffer port map    (en => processor_load,
                                d => clk,
                                q => clk_register);
```

```
RISC : RISC_Processor port map (
                        clk => clk_register,
                        clr => clr,
                        MachineCode => processor_stream,
                        P => P,
                        X0_out => X0_out,
                        X1_out => X1_out,
                        X2_out => X2_out,
                        X3_out => X3_out,
                        X4_out => X4_out,
                        X5_out => X5_out,
                        X6_out => X6_out,
                        X7_out => X7_out,
                        X8_out => X8_out,
                        X9_out => X9_out,
                        X10_out => X10_out,
                        X11_out => X11_out,
                        X12_out => X12_out,
                        X13_out => X13_out,
                        X14_out => X14_out,
                        X15_out => X15_out,
                        X16_out => X16_out,
                        X17_out => X17_out,
                        X18_out => X18_out,
                        X19_out => X19_out,
                        X20_out => X20_out,
                        X21_out => X21_out,
                        X22_out => X22_out,
                        X23_out => X23_out,
                        X24_out => X24_out,
                        X25_out => X25_out,
                        X26_out => X26_out,
                        X27_out => X27_out,
                        X28_out => X28_out,
                        X29_out => X29_out,
                        X30_out => X30_out,
                        X31_out => X31_out
                        );

end Behavioral;
```

8. VGA Terminal “VGA_Terminal.vhd”

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use IEEE.std_logic_unsigned.all;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity text_screen_gen is
    Port ( clk : in STD_LOGIC;
           clr : in STD_LOGIC;
           btn : in STD_LOGIC_VECTOR (2 downto 0);
           ascii : in STD_LOGIC_VECTOR (6 downto 0);
           vidon : in STD_LOGIC;
           hc : in STD_LOGIC_VECTOR (9 downto 0);
           vc : in STD_LOGIC_VECTOR (9 downto 0);
           text_rgb : out STD_LOGIC_VECTOR (2 downto 0));
end text_screen_gen;

architecture Behavioral of text_screen_gen is

component pulse_generator
    Port ( clk : in STD_LOGIC;
           clr : in STD_LOGIC;
           d : in STD_LOGIC;
           q : out STD_LOGIC);
end component;

component fonts
  PORT (
    clka : IN STD_LOGIC;
    addra : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END component;

component tile_memory4096x8
  PORT (
```

```
    a : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
    d : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    dpra : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
    clk : IN STD_LOGIC;
    we : IN STD_LOGIC;
    dpo : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END component;

component block_tile_memory_4096x8
  PORT (
    clka : IN STD_LOGIC;
    wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    addra : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
    dina : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    clkb : IN STD_LOGIC;
    addrb : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
    doutb : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END component;

component nbitregister
  generic(N:integer := 4);
  Port (load : in std_logic;
        clk : in std_logic;
        clr : in std_logic;
        d : in std_logic_vector (N-1 downto 0);
        q : out std_logic_vector (N-1 downto 0));
end component;

    -- font ROM
    signal char_addr : std_logic_vector (6 downto 0);
    signal rom_addr : std_logic_vector (10 downto 0);
    signal row_addr : std_logic_vector (3 downto 0);
    signal bit_addr : std_logic_vector (2 downto 0);
    signal font_word : std_logic_vector (7 downto 0);
    signal font_bit : std_logic;
    -- tile memory
    signal we : std_logic;
    signal addr_r, addr_w : std_logic_vector (11 downto 0);
    signal din, dout : std_logic_vector (6 downto 0);
    -- 80x30 character map
    constant hc_max : integer := 80;
    constant vc_max : integer := 30;
    -- cursor
    signal cur_hc_reg, cur_hc_next : std_logic_vector (6 downto 0);
```

```
   signal cur_vc_reg, cur_vc_next : std_logic_vector (4 downto 0);
   signal btn0_tick, btn1_tick: std_logic;
   signal cursor_on : std_logic;
   -- delayed hc and vc
   signal hc1_reg, vc1_reg : std_logic_vector (9 downto 0);
   signal hc2_reg, vc2_reg : std_logic_vector (9 downto 0);
   -- object output signals
   signal font_rgb, font_rev_rgb : std_logic_vector (2 downto 0);

   -- cursor intermediate signals
   signal dout_sig : std_logic_vector (7 downto 0);
   signal wea : std_logic_vector (0 downto 0);

begin

btn0pulse : pulse_generator port map    (clk => clk,
                                clr => clr,
                                d => btn(0),
                                q => btn0_tick);

btn1pulse : pulse_generator port map    (clk => clk,
                                clr => clr,
                                d => btn(1),
                                q => btn1_tick);

fontrom : fonts port map    (clka => clk,
                        addra => '0' & rom_addr,
                        douta => font_word);

tileram : tile_memory4096x8 port map    (a => addr_w,
                                d => '0' & din,
                                dpra => addr_r,
                                clk => clk,
                                we => we,
                                dpo => dout_sig);

dout <= dout_sig (6 downto 0);

cursorx : nbitregister generic map  (N => 7)
                    port map    (clk => clk,
                                clr => clr,
                                load => '1',
                                d => cur_hc_next,
                                q => cur_hc_reg);

cursory : nbitregister generic map  (N => 5)
```

```
                    port map    (clk => clk,
                                clr => clr,
                                load => '1',
                                d => cur_vc_next,
                                q => cur_vc_reg);

hc1_delay : nbitregister generic map    (N => 10)
                        port map    (clk => clk,
                                    clr => clr,
                                    load => '1',
                                    d => hc,
                                    q => hc1_reg);

hc2_delay : nbitregister generic map    (N => 10)
                        port map    (clk => clk,
                                    clr => clr,
                                    load => '1',
                                    d => hc1_reg,
                                    q => hc2_reg);

vc1_delay : nbitregister generic map    (N => 10)
                        port map    (clk => clk,
                                    clr => clr,
                                    load => '1',
                                    d => vc,
                                    q => vc1_reg);

vc2_delay : nbitregister generic map    (N => 10)
                        port map    (clk => clk,
                                    clr => clr,
                                    load => '1',
                                    d => vc1_reg,
                                    q => vc2_reg);

-- tile memory write
addr_w <= cur_vc_reg & cur_hc_reg;
we <= btn(2);
din <= ascii;

-- tile mrmory read
addr_r <= vc(8 downto 4) & hc(9 downto 3);
char_addr <= dout;

-- font ROM
row_addr <= vc(3 downto 0);
rom_addr <= char_addr & row_addr;
```

```
bit_addr <= hc2_reg(2 downto 0);
font_bit <= font_word(to_integer(unsigned((not bit_addr))));

-- new cursor position
--cur_hc_next <=  (others => '0') when btn0_tick = '1' and cur_hc_reg = (hc_max - 1);
cur_hc_next <=  (others => '0') when btn0_tick = '1' and cur_hc_reg = "1001111" else
              cur_hc_reg + 1 when btn0_tick = '1' else
              cur_hc_reg;
cur_vc_next <=  (others => '0') when btn1_tick = '1' and cur_vc_reg = "11101" else
              cur_vc_reg + 1 when btn1_tick = '1' else
              cur_vc_reg;

-- green over black and reversed for video for cursor, I have no idea what this means
font_rgb <= "111" when font_bit = '1' else
          "000";
font_rev_rgb <= "000" when font_bit = '1' else
          "111";

cursor_on <= '1' when vc2_reg(8 downto 4) = cur_vc_reg and hc2_reg(9 downto 3) =
cur_hc_reg else
          '0';

-- RGB mux circuit
process(vidon, cursor_on, font_rgb, font_rev_rgb)
begin

   if vidon = '0' then
      text_rgb <= "000"; -- blank
   else
      if cursor_on = '1' then
         text_rgb <= font_rev_rgb;
      else
         text_rgb <= font_rgb;
      end if;
   end if;

end process;

end Behavioral;
```

9. Keyboard Serial Communication to ASCII “Keyboard_to_ASCII.vhd”

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Keyboard_to_ASCII is
        port (resetn, clock: in std_logic;
                        ps2c, ps2d: in std_logic;
                        DOUT: out std_logic_vector (7 downto 0);
                        done: out std_logic);
end Keyboard_to_ASCII;

architecture Behavioral of Keyboard_to_ASCII is

component my_ps2keyboard
        port (resetn, clock: in std_logic;
                        ps2c, ps2d: in std_logic;
                        DOUT: out std_logic_vector (7 downto 0);
                        done: out std_logic);
end component;

component ScanCode_to_ASCII_Decoder
   Port (
      ScanCode : in STD_LOGIC_VECTOR (7 downto 0);
      ASCII    : out STD_LOGIC_VECTOR (7 downto 0)
   );
end component;

component pulse_generator
   Port ( clk : in STD_LOGIC;
          clr : in STD_LOGIC;
          d : in STD_LOGIC;
          q : out STD_LOGIC);
end component;

signal done_sig, clr : std_logic;
signal ascii : std_logic_vector (7 downto 0);

begin

clr <= not(resetn);
```

```
keyboard : my_ps2keyboard port map  (resetn => resetn,
                                     clock => clock,
                                     ps2c => ps2c,
                                     ps2d => ps2d,
                                     dout => ascii,
                                     done => done);

decoder : ScanCode_to_ASCII_Decoder port map    (ScanCode => ascii,
                                                 ascii => dout);

end Behavioral;
```

10. RISC Assembler “Assembler_RISC.vhd”

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Assembler_RISC is
    Port ( clk : in STD_LOGIC;
           clr : in STD_LOGIC;
           d : in std_logic_vector (7 downto 0);
           enqueue : in STD_LOGIC;
           enter : in std_logic;
           machine_code : out std_logic_vector (31 downto 0);
           errorstate : out std_logic;
           errortype : out std_logic;
           done : out std_logic
           );
end Assembler_RISC;

architecture Behavioral of Assembler_RISC is

component module_fifo_regs_no_flags
  generic (
    g_WIDTH : natural := 8;
    g_DEPTH : integer := 32
    );
  port (
    i_rst_sync : in std_logic;
    i_clk      : in std_logic;

    -- FIFO Write Interface
    i_wr_en   : in  std_logic;
    i_wr_data : in  std_logic_vector(g_WIDTH-1 downto 0);
    o_full    : out std_logic;

    -- FIFO Read Interface
    i_rd_en   : in  std_logic;
    o_rd_data : out std_logic_vector(g_WIDTH-1 downto 0);
    o_empty   : out std_logic
    );
end component;
```

```
component Assembler_CTRL
   Port ( clk : in STD_LOGIC;
          clr : in STD_LOGIC;
          enter : in STD_LOGIC;
          Instruct_load : out STD_LOGIC_VECTOR (3 downto 0);
          instruction_type : in std_logic_vector (2 downto 0);
          opcode : in std_logic_vector (6 downto 0);
          ascii_in : in std_logic_vector (7 downto 0);
          dequeue : out STD_LOGIC;
          errortype : out std_logic;
          errorstate : out std_logic;

          Rd_vector, Ro_vector, Rs_vector : out std_logic_vector (4 downto 0);
          immed_mem_sel : out std_logic_vector (2 downto 0);
          Immediate_Memory : out std_logic_vector (11 downto 0);
          machine_code_load : out std_logic;
          line_finish : out std_logic;
          prom_code_load : out std_logic
          );
end component;

component nbitregister
  generic(N:integer);
  Port (load : in std_logic;
        clk : in std_logic;
        clr : in std_logic;
        d : in std_logic_vector (N-1 downto 0);
        q : out std_logic_vector (N-1 downto 0));
end component;

component ASCIItoOpcodeDecoder
   Port ( ASCII : in STD_LOGIC_VECTOR (31 downto 0);
          opcode : out std_logic_vector (6 downto 0));
end component;

component Instruction_Type_Decoder
   Port ( opcode : in STD_LOGIC_VECTOR (6 downto 0);
          instruction_type : out STD_LOGIC_VECTOR (2 downto 0));
end component;

constant width : natural := 8;
constant depth : integer := 80;

-- Signals
signal dequeue, machine_code_load, line_finish, CTRL_clr, prom_code_load : std_logic;
signal instruction_type, immed_mem_sel : std_logic_vector (2 downto 0);
signal instruct_load : std_logic_vector (3 downto 0);
signal Rd_vector, Ro_vector, Rs_vector : std_logic_vector (4 downto 0);
signal opcode : std_logic_vector (6 downto 0);
signal queueout : std_logic_vector (7 downto 0);
```

```
signal Immediate_Memory : std_logic_vector (11 downto 0);
signal instructions, decoderin, machine_code_sig : std_logic_vector (31 downto 0);

begin

queue : module_fifo_regs_no_flags generic map   (g_width => width,
                                    g_depth => depth)
                            port map    (i_rst_sync => CTRL_CLR,
                                    i_clk => clk,
                                    i_wr_en => enqueue,
                                    i_wr_data => d,
                                    o_full => open,
                                    i_rd_en => dequeue,
                                    o_rd_data => queueout,
                                    o_empty => open);

CTRL : Assembler_CTRL port map  (clk => clk,
                        clr => CTRL_CLR,
                        enter => enter,
                        Instruct_load => instruct_load,
                        instruction_type => instruction_type,
                        ascii_in => queueout,
                        dequeue => dequeue,
                        opcode => opcode,
                        errortype => errortype,
                        errorstate => errorstate,
                        Rd_vector => Rd_vector,
                        Ro_vector => Ro_vector,
                        Rs_vector => Rs_vector,
                        immed_mem_sel => immed_mem_sel,
                        Immediate_Memory => Immediate_Memory,
                        machine_code_load => machine_code_load,
                        line_finish => line_finish,
                        prom_code_load => prom_code_load);

CTRL_CLR <= clr or line_finish;
done <= prom_code_load;

instruct1reg : nbitregister generic map (N => 8)
                    port map    (clk => clk,
                                clr => CTRL_CLR,
                                d => queueout,
                                q => instructions(31 downto 24),
                                load => instruct_load(0));

instruct2reg : nbitregister generic map (N => 8)
                    port map    (clk => clk,
                                clr => CTRL_CLR,
                                d => queueout,
                                q => instructions(23 downto 16),
```

```
                        load => instruct_load(1));

instruct3reg : nbitregister generic map (N => 8)
                port map    (clk => clk,
                        clr => CTRL_CLR,
                        d => queueout,
                        q => instructions(15 downto 8),
                        load => instruct_load(2));

instruct4reg : nbitregister generic map (N => 8)
                port map    (clk => clk,
                        clr => CTRL_CLR,
                        d => queueout,
                        q => instructions(7 downto 0),
                        load => instruct_load(3));

asciitoopcode_decoder : ASCIItoOpcodeDecoder port map   (ASCII => instructions,
                                opcode => opcode);

instructiontype_decoder : instruction_type_decoder port map (opcode => opcode,
                                instruction_type => instruction_type);

machine_code_sig <= "0000000" & Ro_vector & Rs_vector & "000" & Rd_vector & opcode when
instruction_type = "000" else
        Immediate_Memory & Rs_vector & immed_mem_sel & Rd_vector & opcode when
instruction_type = "001" else
        Immediate_Memory & Ro_vector & "000" & "11111" & opcode when instruction_type =
"010" else
        Immediate_Memory & "11111" & "001" & "11111" & opcode when instruction_type =
"011" else
        x"00000000";

machine_code_reg : nbitregister generic map (N => 32)
                port map    (clk => clk,
                        clr => clr,
                        d => machine_code_sig,
                        q => machine_code,
                        load => machine_code_load);

end Behavioral;
```