Lab 8 Report

Matthew Crump A01841001
Nicholas Williams A02057223

Objectives

Add 3 DLX instructions to the assembler and DLX architecture. Instantiate the DLX processor on the Intel DE10-Lite development board. Print characters, integers, and unsigned integers to the screen via UART.

Procedure

We started this lab by first altering our assembler to be able to handle the three new instructions PCH (print character to the screen via UART), PD (print signed decimal integer to screen via UART), and PDU (print unsigned decimal integer to screen via UART). Our assembler not only creates the appropriate files, but it also checks for proper grammar and spelling in the assembly file. This meant that we had to improve the grammar and spell checker to handle the three new instructions as well. We also updated our Python DLX simulator to allow for these new instructions. This will aid us in debugging later.

We then went through the VHDL that we had from Lab 7 and created a top module and wrapper module that implements the modules from Lab 7 along with the required modules for Lab 8. The wrapper module connects all of the signals between modules, and the top module connects the pins, clk, and reset to the wrapper. We did it this way so that our testbench can also connect to the wrapper module just like the top module typically does. We then pulled in our UART module from Lab 1 and connected its signals. We disconnected the receive pin from the UART, so we will have to connect that signal before we implement the scan modules in the future.

We then made a print module that watched for 1 of the three new opcodes to enter the execute stage. The vhdl file for the print module also connected the signals for the FIFO, LIFO, DIVIDER, and UART. As soon as one of the three new opcodes is found the corresponding register value is thrown into the FIFO to act as a buffer. We then created a finite state machine that handles the registered value accordingly based on whether it is a character, unsigned decimal, or decimal value (see Figure 1). The finite state machine sends the signed and unsigned decimal values through the divider and remainder bits into the LIFO. We then set a flag that tells the UART to start pulling from

the LIFO, and we wait until the LIFO is empty to go back to an IDLE state. From here, we have the potential to pull another value from the FIFO and start the process all over again.

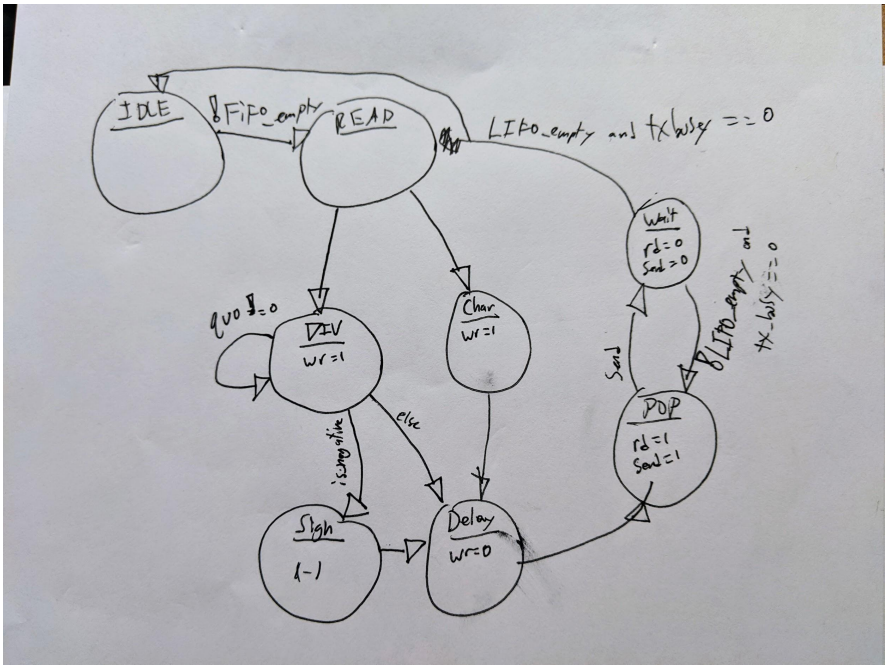

Figure 1. Finite state machine for print module

Some additional modifications needed to happen in the decode and execute stages. We had originally typed some conditions as being just greater than or equal to the jump instruction, but were not bound on the upper end. Without these changes, it could have possibly caused a stall condition.

Finally, the last thing we needed to do was get the DLX factorial program to use the new print statements. We typed out the “Welcome to the DLX factorial program!” string then one helper string of “! = ”. We then created small loops that would print each string character by character. This also included print unsigned decimal instructions for our input and output of the factorial program.

Results

We wrote our modules and then worked through the errors until the project successfully compiled. We ran into errors with the divider component and had to look up how the divider is properly instantiated. We decided to program the DE10-Lite board with the compiled build, but to no surprise it did not work. We then wrote a testbench to see which signals were failing to behave correctly.

Through simulation we found out that our handshake with the LIFO and UART was not working as desired and it sent write/read signals that lasted for 2 clock cycles each, instead of one. We modified our original state machine to use one additional state to create more time between reading out the LIFO and saving the byte in the UART's send buffer. We then got data on the TX line, however it was all zeros. This was just an off-by-one error when sending data from FIFO and into the LIFO when setting the write enable registers.

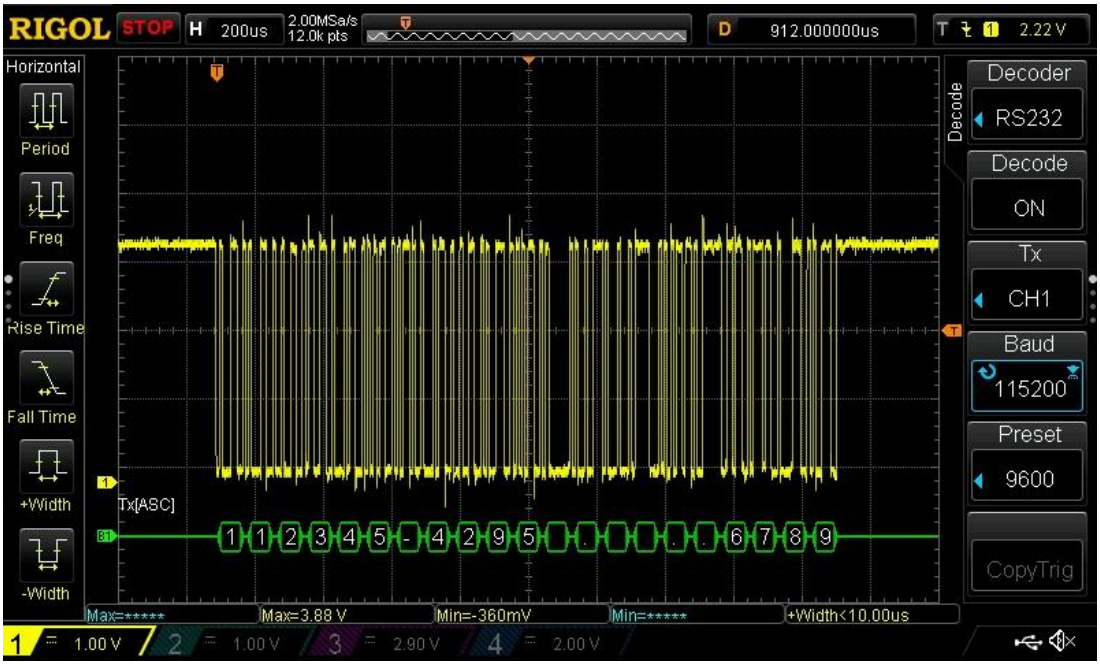

Figure 2. Oscilloscope display of transmitted signal

We fixed the problem and got ASCII values on the TX line, but there was a zero at the start of each printed number. We realized that the LIFO was being written to one too many times. We quickly fixed the mistake by making the state machine disable the LIFO write enable one clock cycle earlier. One thing we forgot to make in the design stage of this lab was to make negative numbers be inserted into the divider in the two's complement form, else -32k would be interpreted as negative 4 billion. A simple mux to switch between these two values was all that was needed.

We moved on from a simple test program and tried our new factorial program with print statements. The output had many repeated characters after the main welcome string was printed. We caused this by inserting data into the FIFO during a stall condition which then caused the instruction to be repeated. We fixed this by adding the stall signal as a condition for inserting into the FIFO. Now, the output was closer to what we expected, but the first character was null and the last character in the string was skipped. It turned out the data forwarding was not working so we got the character from

the previous loop iteration which explained why the first character was always wrong and the last character was always skipped. We actually corrected this one in software by making our DLX assembler put the source register in the RS1 spot rather than the RS2 or RD place. This fixed the problem, but then sometimes it would not print. The instructions were somehow interpreted as a stall condition. We fixed this in software by setting the RD field of the instruction, although unused, to 31. This made it so that the check for RS1 equal to RD (along with some other conditions), would not cause a stall.

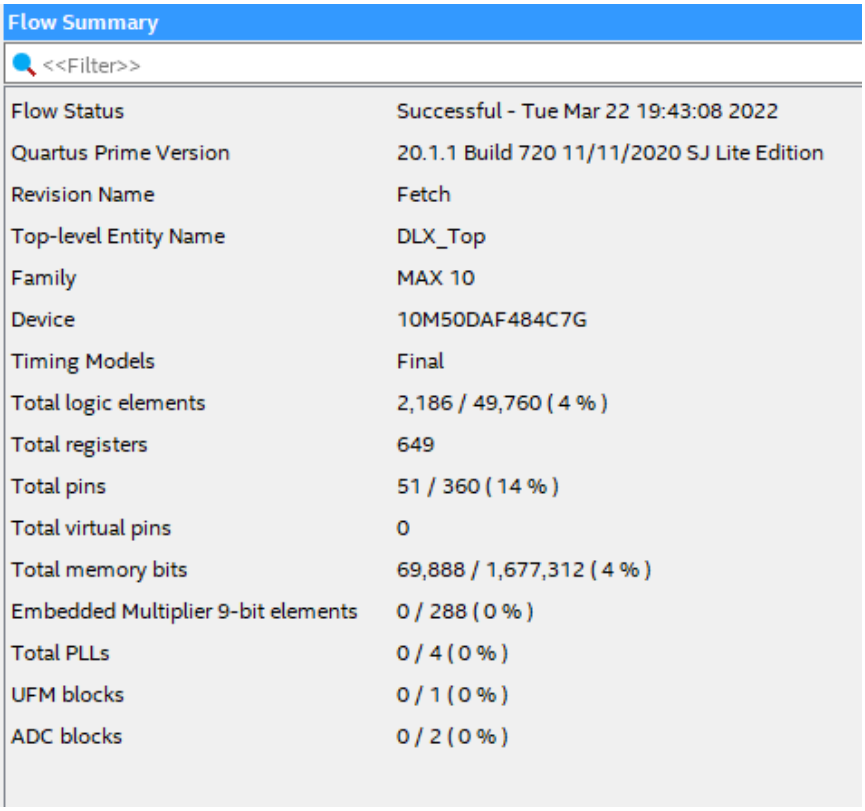

Figure 3. Implementation Results

We finally got everything on the board and working properly. Figure 4 shows the serial terminal output for our working lab. We will admit that it is not as well tested as we would like it to be and we most definitely need to worry about fixing timing issues, but we are happy with the results we have achieved thus far.

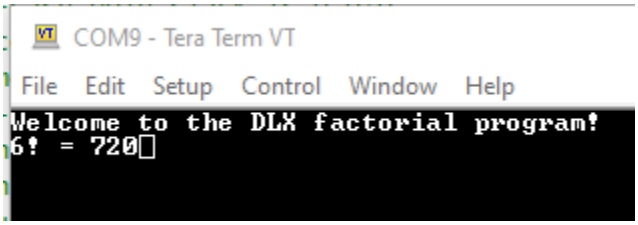

Figure 4. Final output to the serial terminal

Conclusion

Overall we are very pleased with our project's ability to handle the three new instructions PCH, PD, and PDU. We eventually got the output to be what it needed to be for this lab, but we will admit that our processor is certainly not bug free. The good news is that our CPU is still calculating 6 factorial correctly, and we are able to print it to the screen. Hopefully before the final project we can fix all of the bugs in our code to truly handle all situations we could encounter on the final project. As shown in the oscilloscope output, if our program is given a long number to translate, the divider can't keep up at this clock speed and spits out garbage half way through. Soon, we will hook up the print module to the 10 MHz clock to allow the divider enough time to do its thing.

Appendix A

```
library ieee, lpm, work;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
use lpm.lpm_components.all;
use work.dlx_package.all;

entity DLX_Print_Scan is
    generic (
        g_UART_WIDTH : natural := 8;
        g_TX_DEPTH : natural := 16;
        g_DEPTH : natural := 64;
        g_FIFO_WIDTH : natural := 34
    );
    port
    (
        clk : in std_logic;
        rstn : in std_logic;
        invalid : in std_logic;
        print_data : in std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
        op_code : in std_logic_vector(c_DLX_OPCODE_WIDTH-1 downto 0);
        uart_rx : in std_logic;
        tx_busy : out std_logic;
        uart_tx : out std_logic
    );

    end entity;

architecture rtl of DLX_Print_Scan is
    component LPM_DIVIDE
              generic (LPM_WIDTHN : natural;
                       LPM_WIDTHD : natural;
    LPM_NREPRESENTATION : string := "UNSIGNED";
    LPM_DREPRESENTATION : string := "UNSIGNED";
    LPM_PIPELINE : natural := 0;
    LPM_TYPE : string := L_DIVIDE;
    LPM_HINT : string := "UNUSED");
    port (NUMER : in std_logic_vector(LPM_WIDTHN-1 downto 0);
    DENOM : in std_logic_vector(LPM_WIDTHD-1 downto 0);
    ACLR : in std_logic := '0';
    CLOCK : in std_logic := '0';
    CLKEN : in std_logic := '1';
    QUOTIENT : out std_logic_vector(LPM_WIDTHN-1 downto 0);
    REMAIN : out std_logic_vector(LPM_WIDTHD-1 downto 0));
    end component;

    -- 2's complelent
    signal twos_complement : std_logic_vector(c_DLX_WORD_WIDTH-1 downto
0);
    signal data_in : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    -- divider
```

```
    signal div_input : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    signal quotient : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    signal cmd : std_logic_vector(1 downto 0);
    signal is_negative : std_logic;
    signal sign_bit : std_logic;
    signal div_data : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);

    -- FIFO
    -- write
    signal fifo_full : std_logic;
    signal fifo_wr_en : std_logic;
    -- read
    signal fifo_empty : std_logic;
    signal fifo_rd_en : std_logic;
    signal fifo_rd_data : std_logic_vector(g_FIFO_WIDTH-1 downto 0);
    signal fifo_wr_data : std_logic_vector(g_FIFO_WIDTH-1 downto 0);

    -- LIFO
    -- write
    signal lifo_full : std_logic;
    signal lifo_wr_en : std_logic;
    signal lifo_wr_data : std_logic_vector(g_UART_WIDTH-1 downto 0);
    signal div_char : std_logic_vector(g_UART_WIDTH-1 downto 0);
    -- read
    signal lifo_empty : std_logic;
    signal lifo_rd_en : std_logic;

    -- UART
    signal rx_data_valid : std_logic;
    signal tx_data_valid : std_logic;
    signal tx_byte : std_logic_vector(g_UART_WIDTH-1 downto 0);
    signal received_byte : std_logic_vector(g_UART_WIDTH-1 downto 0);

    -- State Machine
    type state_type is (s_IDLE, s_READ, s_CHAR, s_SIGNED, s_DIVIDE,
s_DELAY, s_POP, s_WAIT);
    signal print_state : state_type;
    signal lifo_in_sel : std_logic;
    signal tx_ready : std_logic;

begin
    twos_complement <= (not fifo_rd_data(31 downto 0)) + '1';
    data_in <= twos_complement when (fifo_rd_data(33 downto 32) =
c_DLX_PD(1 downto 0) and fifo_rd_data(3) = '1') else fifo_rd_data(31
downto 0);
    process(clk)
    begin
        if(rising_edge(clk)) then
            if op_code >= c_DLX_PCH and op_code <= c_DLX_PDU and invalid =
'0' then
                fifo_wr_data <= op_code(1 downto 0) & print_data;
                fifo_wr_en <= not fifo_full;
            else
                fifo_wr_en <= '0';
```

```
            end if;
        end if;
    end process;

    process(clk)
    begin
        if(rising_edge(clk)) then
            case print_state is
                when s_IDLE =>
                    if fifo_empty = '0' then
                        fifo_rd_en <= '1';
                        div_data <= data_in;
                        sign_bit <= fifo_rd_data(31);
                        cmd <= fifo_rd_data(33 downto 32);
                        print_state <= s_READ;
                    else
                        fifo_rd_en <= '0';
                        print_state <= s_IDLE;
                    end if;

                when s_READ =>
                    fifo_rd_en <= '0';
                    if cmd = c_DLX_PCH(1 downto 0) then
                        print_state <= s_CHAR;
                    else
                        print_state <= s_DIVIDE;
                        lifo_wr_en <= '1';
                        if cmd = c_DLX_PD(1 downto 0) then
                            is_negative <= sign_bit;
                        else
                            is_negative <= '0';
                        end if;
                    end if;

                when s_CHAR =>
                    print_state <= s_DELAY;
                    lifo_in_sel <= '1';
                    lifo_wr_en <= '1';

                when s_SIGNED =>
                    lifo_wr_en <= '0';
                    print_state <= s_DELAY;

                when s_DIVIDE =>
                    if quotient = x"00000000" then
                        if is_negative = '1' then
                            print_state <= s_SIGNED;
                        else
                            lifo_wr_en <= '0';
                            print_state <= s_DELAY;
                        end if;
                    else
                        lifo_wr_en <= '1';
                        print_state <= s_DIVIDE;
```

```
                end if;

            when s_DELAY =>
                lifo_wr_en <= '0';
                lifo_in_sel <= '0';
                print_state <= s_POP;

            when s_POP =>
                lifo_rd_en <= '1';
                tx_ready <= '0';
                if lifo_empty = '1' and tx_busy = '0' then
                    print_state <= s_IDLE;
                else
                    print_state <= s_WAIT;
                end if;

            when s_WAIT =>
                lifo_rd_en <= '0';
                tx_ready <= '1';
                if lifo_empty = '0' and tx_busy = '0' then
                    print_state <= s_POP;
                elsif lifo_empty = '1' and tx_busy = '0' then
                    tx_ready <= '0';
                    print_state <= s_IDLE;
                end if;

            when others =>
                print_state <= s_IDLE;
        end case;
    end if;
end process;

PRINT_BUF: entity work.FIFO(rtl)
    generic map (
        g_WIDTH => g_FIFO_WIDTH,
        g_DEPTH => g_DEPTH
    )
    port map (
        clk => clk,
        rstn => rstn,
        full => fifo_full,
        wr_en => fifo_wr_en,
        wr_data => fifo_wr_data,
        empty => fifo_empty,
        rd_en => fifo_rd_en,
        rd_data => fifo_rd_data
);

div_input <= div_data when fifo_rd_en = '1' else quotient;

DIV: component LPM_DIVIDE
generic map (
    LPM_WIDTHN => c_DLX_WORD_WIDTH,
    LPM_WIDTHD => g_UART_WIDTH,
```

```
        LPM_PIPELINE => 1
    )
    port map (
        clock => clk,
        clken => '1',
        aclr => not rstn,
        numer => div_input,
        denom => std_logic_vector(to_unsigned(10, g_UART_WIDTH)),
        quotient => quotient,
        remain => div_char
    );

    lifo_wr_data <= div_data(7 downto 0) when lifo_in_sel = '1' else x"2D"
when (print_state = s_SIGNED) else div_char + x"30";

    TX_BUF: entity work.LIFO(rtl)
    generic map (
        g_WIDTH => g_UART_WIDTH,
        g_DEPTH => g_TX_DEPTH
    )
    port map (
        clk => clk,
        rstn => rstn,
        full => lifo_full,
        wr_en => lifo_wr_en,
        wr_data => lifo_wr_data,
        empty => lifo_empty,
        rd_en => lifo_rd_en,
        rd_data => tx_byte
    );

    tx_data_valid <= tx_ready;

    UART1: entity work.uart(rtl)
    generic map (
        -- clk_freq / BAUD = clks_per_bit
        -- 50MHz / 19200 = 2604
        clks_per_bit => 434
    )
    port map (
        clk => clk,
        -- rx
        rx_serial => uart_rx,
        rx_data_valid => rx_data_valid,
        rx_byte => received_byte,
        -- tx
        tx_serial => uart_tx,
        tx_data_valid =>tx_data_valid,
        tx_byte => tx_byte,
        tx_busy => tx_busy
    );

end rtl;
```