Lab 9 Report

Matthew Crump A01841001
Nicholas Williams A02057223

Objectives

The purpose of this lab was to develop the VHDL required to properly scan decimal and unsigned decimal values from a serial terminal. In order to scan values from a terminal, additional VHDL was needed to stall the pipeline and convert strings to decimal values. The goal is to make the DLX processor become more like a useful computer by properly handling input and output from a user.

Procedure

We started out by updating our DLX assembler to include the new instruction. We then updated our dlx assembly program to have the new scan instruction in it. We then looked at the lecture slides on how to turn character arrays into integers. This helped us decide how we wanted to lay out the logic for our scan module.

We ended up using one FIFO with a depth of 16 that then fed into a multiplier and adder. The UART receiver subtracted 48 from the received character and then pushed the result into the FIFO to save it from having to do that in the next stage. The state machine waited until the new line and carriage return characters were received (see Figure 1). Once they were received, the FIFO emptied and put the string of integers through the multiplier and adder circuits. The result was then saved in an accumulator register which was then fed back into the multiplier circuit. Once the FIFO was empty, the data valid signal was set high to let the CPU know it had gotten an integer from the UART.

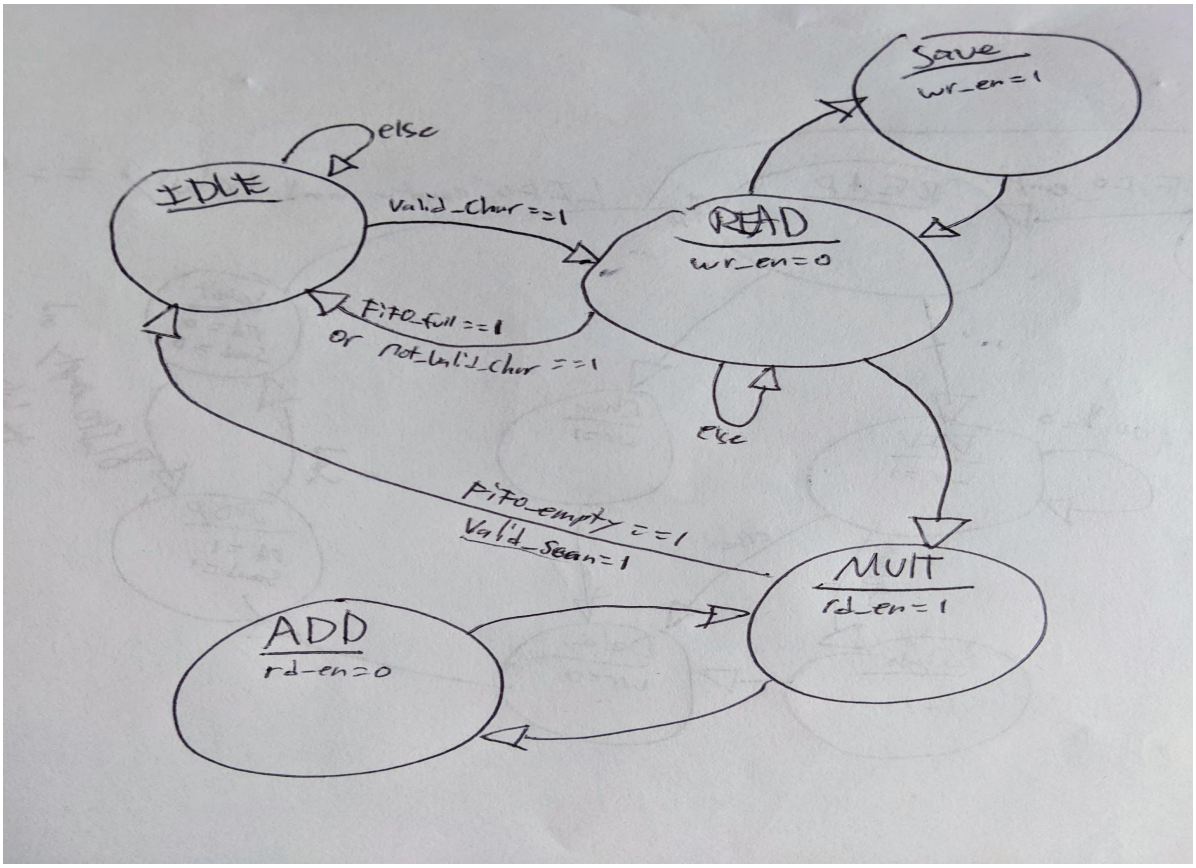


Figure 1. Finite state machine for scan module

We added the GD and GDU instructions to the instruction list that will stall the pipeline when received. The stall is indefinite until a valid scan event occurs. The data then flows into a mux that switches out Operand_0's input with the ALU. The data then makes its way back through to the CPU registers like all other instructions. Finally, we made the GD and GDU instructions set the writeback enable signal.

To clean up our code and make functionality a bit better, we completed some housekeeping items. We moved the print and scan logic into separate files because we did not want name conflicts in signals. We figured that it would be a nightmare to keep all the signals straight with so many FIFOs. We modified the assembler to allow for escape characters so that we could include newlines and carriage returns in the DLX strings. Finally, we added a PLL to generate two clocks for our DLX CPU. We wanted a 50 MHz clock for the processor itself and a 20 MHz clock for the IO print and scan modules. We wanted to get the multiplier and divider to meet timing without slowing down the entire processor to that same speed.

Results

Our first design for the state machine almost worked, but it performed one extra multiply at the end. We tested out the C program by reordering the 2 statements to see if they were dependent on the order. We obtained similar results to the simulation. To solve this, we developed an additional state to multiply and then add, just like the C program. Rerunning the simulation, we got the correct results.

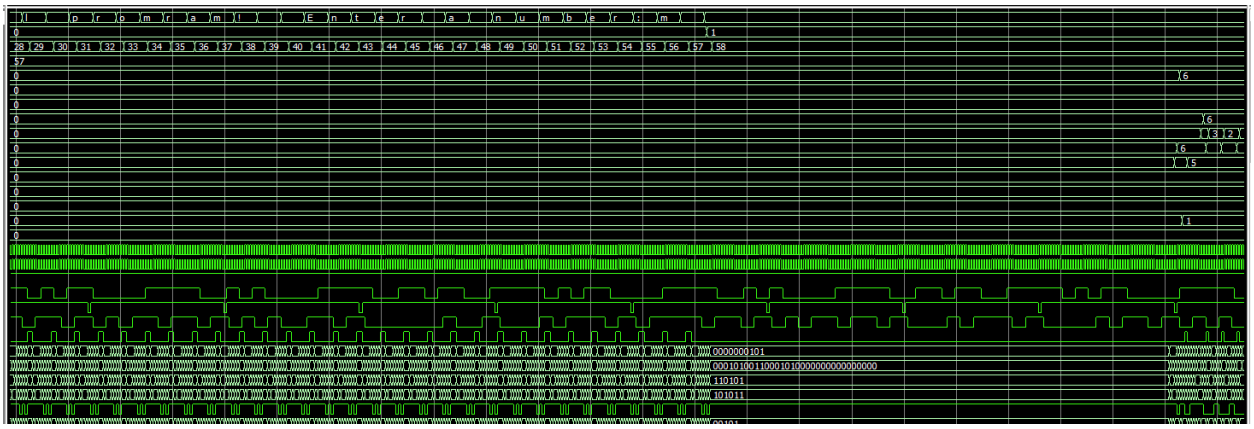
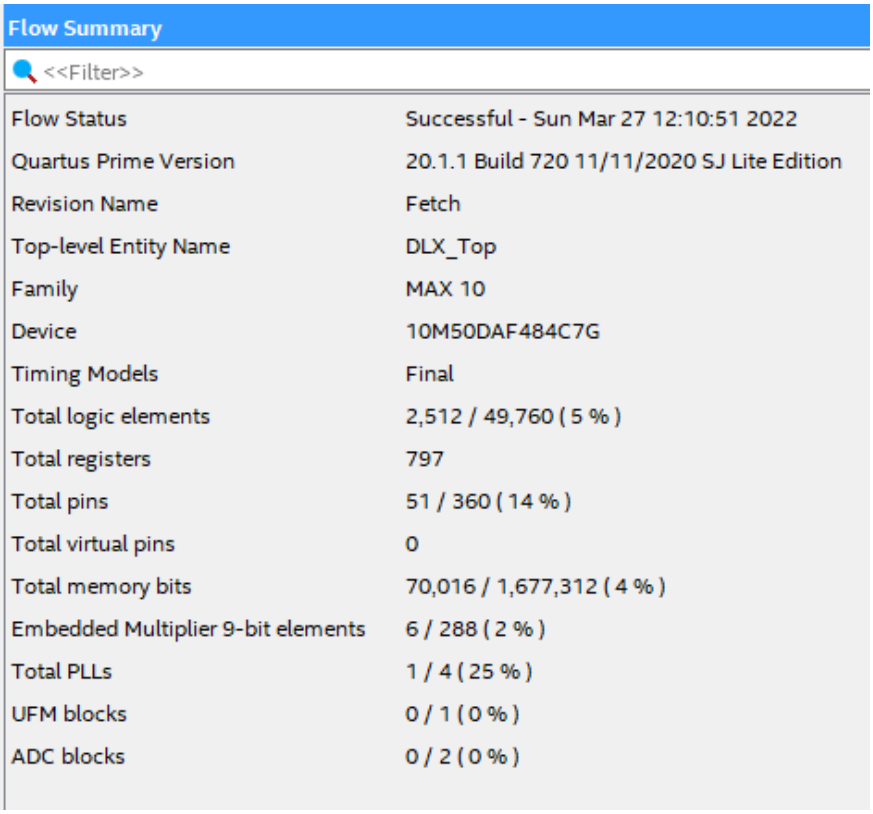


Figure 2. Factorial simulation with pipeline stall

Surprisingly, the pipeline stall worked on the first try. However, it did not get the data fast forwarded into the correct register in time. Originally, we had the pipeline stall as the instruction left the execute stage. We then moved the logic to stall in the beginning of the execute stage and that produced the correct results.

Adding a second clock worked on the first try, but we thought it was not working right because the terminal output was just garbage. We discovered that a generic that sets the clocks per bit was set to the wrong value in the top level VHDL file. We had set the generic to the correct value in the file below. but it was overwritten by the top file.



Flow Summary

<<Filter>>

Flow Status	Successful - Sun Mar 27 12:10:51 2022
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	Fetch
Top-level Entity Name	DLX_Top
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	2,512 / 49,760 (5 %)
Total registers	797
Total pins	51 / 360 (14 %)
Total virtual pins	0
Total memory bits	70,016 / 1,677,312 (4 %)
Embedded Multiplier 9-bit elements	6 / 288 (2 %)
Total PLLs	1 / 4 (25 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 3. Implementation Results

Figure 3 shows the implementation results for lab 9. We increased the amount of logic elements and registers from lab 8. We are also now using 6 of the embedded multiplier elements. Figure 4 shows the factorial example first asking for user input and then displaying the result in the user terminal.

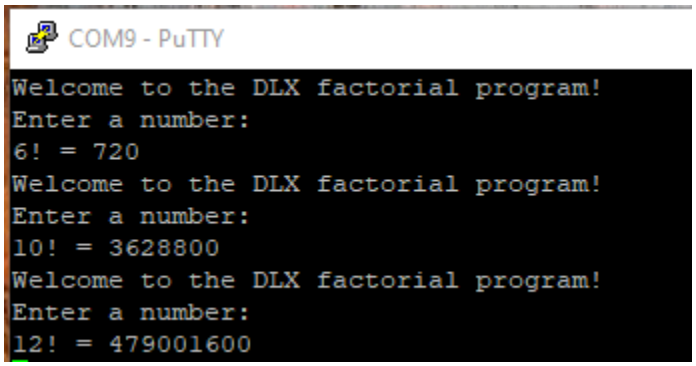


Figure 4. Final output to the serial terminal

Conclusion

Overall, we are very pleased with how simple this lab was to implement. We had a few bugs to work through, but no major issues. We are even more pleased that our DLX processor can now receive input from a user. After finishing this lab, we decided to start discussing how we can improve timing so that we can win the competition for the final lab. We added a PLL so that we can run the CPU and scan/print logic at different frequencies to increase the execution speed.

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_Scan is
    generic (
        g_DEPTH : natural := 16;
        g_UART_WIDTH : natural := 8
    );
    port
    (
        clk : in std_logic;
        clk_io  : in std_logic;
        rstn : in std_logic;
        rx_data_valid : in std_logic;
        rx_byte : in std_logic_vector(g_UART_WIDTH-1 downto 0);
        scan_valid : out std_logic;
        scan_data : out std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0)
    );

    end entity;

architecture rtl of DLX_Scan is

    component LPM_MULT
        generic ( LPM_WIDTHA : natural;
            LPM_WIDTHB : natural;
            LPM_WIDTHS : natural := 1;
            LPM_WIDTHP : natural;
            LPM_REPRESENTATION : string := "UNSIGNED";
            LPM_PIPELINE : natural := 0;
            LPM_TYPE: string := L_MULT;
            LPM_HINT : string := "UNUSED");
        port ( DATAA : in std_logic_vector(LPM_WIDTHA-1 downto 0);
            DATAB : in std_logic_vector(LPM_WIDTHB-1 downto 0);
            ACLR : in std_logic := '0';
            CLOCK : in std_logic := '0';
            CLKEN : in std_logic := '1';
            SUM : in std_logic_vector(LPM_WIDTHS-1 downto 0) := (OTHERS => '0');
            RESULT : out std_logic_vector(LPM_WIDTHP-1 downto 0));
    end component;

    -- multiplyer
    signal mult_result : std_logic_vector((c_DLX_WORD_WIDTH * 2)-1 downto 0);
    signal mult_accum : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);


    -- 2's complelent
    signal twos_complement : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    signal data_out : 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_UART_WIDTH-1 downto 0);
    signal fifo_wr_data : std_logic_vector(g_UART_WIDTH-1 downto 0);
```

```
    -- State Machine
    type state_type is (s_IDLE, s_READ, s_SAVE, s_MULT, s_ADD);
    signal scan_state : state_type;
    signal mult_en : std_logic;
    signal is_negative : std_logic;
    signal number_char : std_logic;
    signal escape_char : std_logic;

    signal is_valid : std_logic;
    signal is_valid_dly : std_logic;

begin

    twos_complement <= (not mult_accum) + '1';
    data_out <= mult_accum when is_negative = '0' else twos_complement;

    scan_valid <= is_valid and not is_valid_dly;
    process(clk)
    begin
        if(rising_edge(clk)) then
            is_valid_dly <= is_valid;
        end if;
    end process;
    process(clk_io)
    begin
        if(rising_edge(clk_io)) then
            if mult_en = '1' then
                mult_accum <= mult_result(c_DLX_WORD_WIDTH-1 downto 0) + fifo_rd_data;
            else
                mult_accum <= (others => '0');
            end if;
            scan_data <= data_out;
        end if;
    end process;

    number_char <= '1' when rx_byte >= x"30" and rx_byte <= x"39" else '0';
    escape_char <= '1' when rx_byte = x"0A" or rx_byte = x"0D" else '0';

    process(clk_io)
    begin
        if(rising_edge(clk_io)) then
            case scan_state is
                when s_IDLE =>
                    if rx_data_valid = '1' then
                        if rx_byte = x"2D" then
                            is_negative <= '1';
                            scan_state <= s_READ;
                        elsif number_char = '1' then
                            scan_state <= s_READ;
                            fifo_wr_en <= '1';
                        end if;
                    else
                        scan_state <= s_IDLE;
                        is_negative <= '0';
                        is_valid <= '0';
                        mult_en <= '0';
                        fifo_rd_en <= '0';
                        fifo_wr_en <= '0';
                    end if;

                when s_READ =>
                    if escape_char = '1' then
                        scan_state <= s_MULT;
                        mult_en <= '1';
                    elsif rx_data_valid = '1' then
                        if number_char = '1' then
```

```
                        fifo_wr_en <= '1';
                        scan_state <= s_SAVE;
                    elsif (number_char = '0' or fifo_full = '1') then
                        scan_state <= s_IDLE;
                    end if;
                else
                    fifo_wr_en <= '0';
                    scan_state <= s_READ;
                end if;

            when s_SAVE =>
                scan_state <= s_READ;
                fifo_wr_en <= '0';

            when s_MULT =>
                fifo_rd_en <= '1';
                if fifo_empty = '1' then
                    is_valid <= '1';
                    scan_state <= s_IDLE;
                else
                    scan_state <= s_ADD;
                end if;

            when s_ADD =>
                scan_state <= s_MULT;
                fifo_rd_en <= '0';

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

fifo_wr_data <= rx_byte - x"30";
SCAN_BUF: entity work.FIFO(rtl)
    generic map (
        g_WIDTH => g_UART_WIDTH,
        g_DEPTH => g_DEPTH
    )
    port map (
        clk => clk_io,
        rstn => rstn,
        full => fifo_full,
        wr_en => fifo_wr_en,
        wr_data => fifo_wr_data,
        empty => fifo_empty,
        rd_en => fifo_rd_en and not fifo_empty,
        rd_data => fifo_rd_data
);

MUL: component LPM_MULT
    generic map (
        LPM_WIDTHA => c_DLX_WORD_WIDTH,
        LPM_WIDTHB => c_DLX_WORD_WIDTH,
        LPM_WIDTHP => c_DLX_WORD_WIDTH * 2,
        LPM_PIPELINE => 1
    )
    port map (
        clock => clk_io,
        clken => '1',
        aclr => not rstn,
        dataa => x"0000000A",
        datab => mult_accum,
        result => mult_result
    );

end rtl;
```

[redacted]