Lab 7 Report

Matthew Crump A01841001
Nicholas Williams A02057223

Objectives

Build a functional DLX CPU that does not require any strategically placed NOPs to function correctly. Learn under what conditions data fastforwarding and stalls happen then implement control logic to handle this.

Procedure

We broke up the design goals into four main tasks: data hazards for ALU operations, data hazards for load instructions, control hazards for jumps and control hazards for branches. We refactored much of the code (by renaming the ports) to make it much easier to remember where each signal was going. We also switched the synthesis and simulation tools to use VHDL 2008 as this allowed output signals to be read inside of the modules, which reduced the number of “duplicate” signals we needed to write.

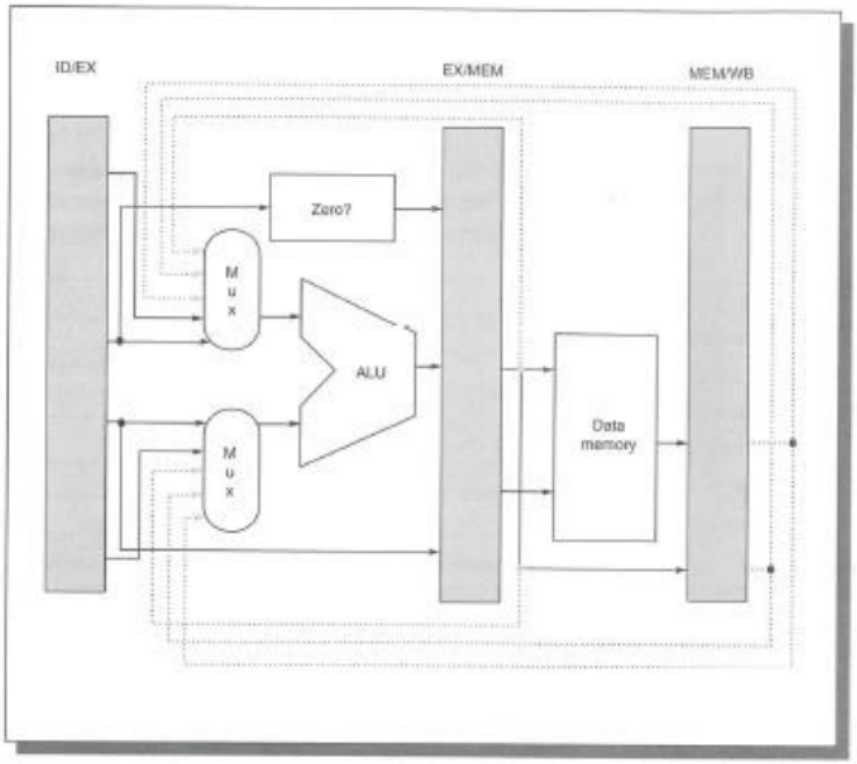


Figure 1. DLX Data/Control Hazard Diagram

To create a benchmark of sorts, we timed our lab 6 DLX CPU which has no data/control hazards to know how much of an improvement we made by adding in that logic (see Figure 2). We also modified our Python DLX simulator to count how many times each branch instruction was taken or not taken in order to know how we wanted to handle branch instructions. We needed to either assume they would always be taken or not always be taken. This also provided a way for us to potentially do code analysis and have the assembler modify the branch instructions with a “prediction bit” that could be used to switch between the two cases.

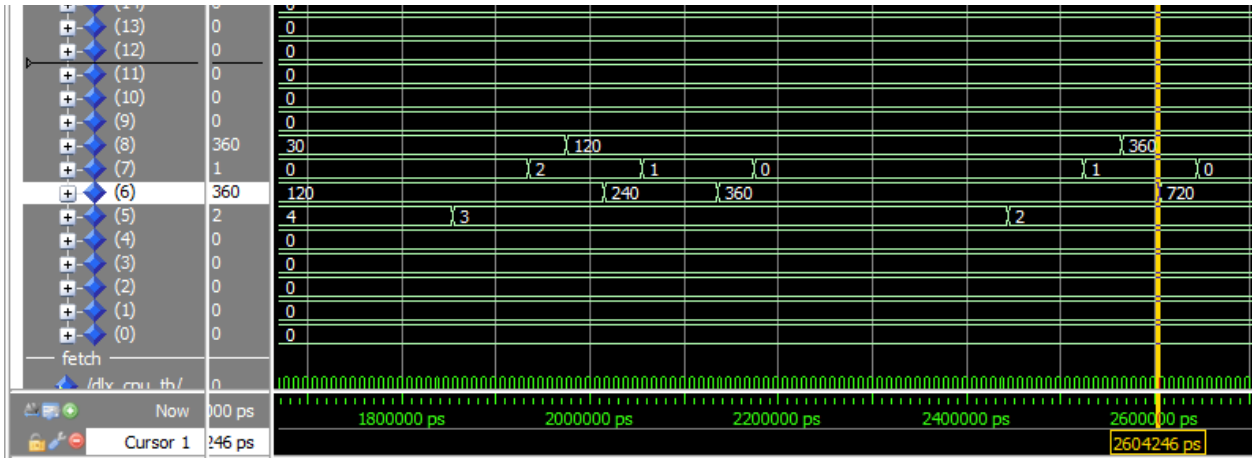


Figure 2. Six Factorial Benchmark With NOPs

We started with the decode stage by forwarding the RS1 and RS2 values onto the execute stage. These registers were also later forwarded onto the writeback stage. RS1 and RS2 were then backtracked into the execute stage to allow the control logic to make the correct decision when forwarding data. With that in place we could start using the information given on the data/control hazard rule table to create the necessary logic. The first 4 data hazards were implemented in less than an hour as they were rather trivial. We then modified our NOP factorial program to not have NOPs in-between ALU operations for testing purposes.

We then made the pipeline stall when a register operation was preceded by a load instruction. We modified the pipeline registers in the fetch, decode, and execute stages to loop back the previous values if a stall condition was detected. The writeback did not need this as it just reran the last valid command which has no negative effects on the program. The PC also had to be altered in a stall condition to be PC minus one, because in a stall condition the PC was already incremented by one and thus must be decremented again to point to the correct instruction.

The last part was to get the jump and branch instructions to work without NOPs. For jump instructions, the decode stage marked the next 2 instructions as invalid while the new jump address got fed into the execute-fetch boundary. Thus, the invalid instructions after the jump did not get executed and no NOPs would be needed in between jump instructions. Similar logic followed for the branch instructions, but rather than invalidating them in the decode stage, that logic happened in the execute stage. If a branch was not taken, nothing different happened in the CPU. If the branch was taken, then the next three instructions were invalidated which caused them to not have any effect at all.

Results

Using our NOP benchmark for comparison, we got a time of 2,600 ns. With the data/control hazard logic we achieved a time of 1,230 ns. Making a difference of 1,370 ns which is more than half the time! As expected, implementing data and control hazards is a worthwhile enhancement.

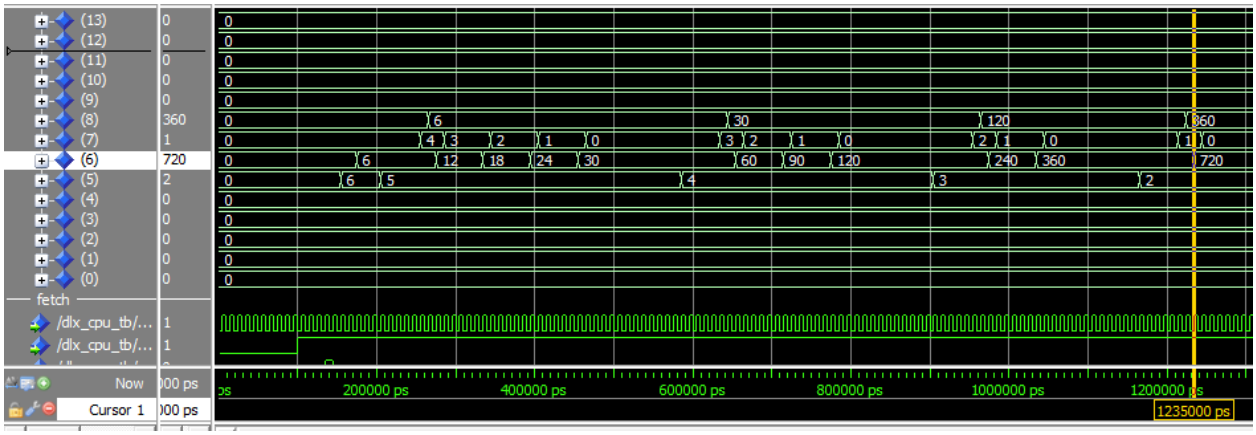
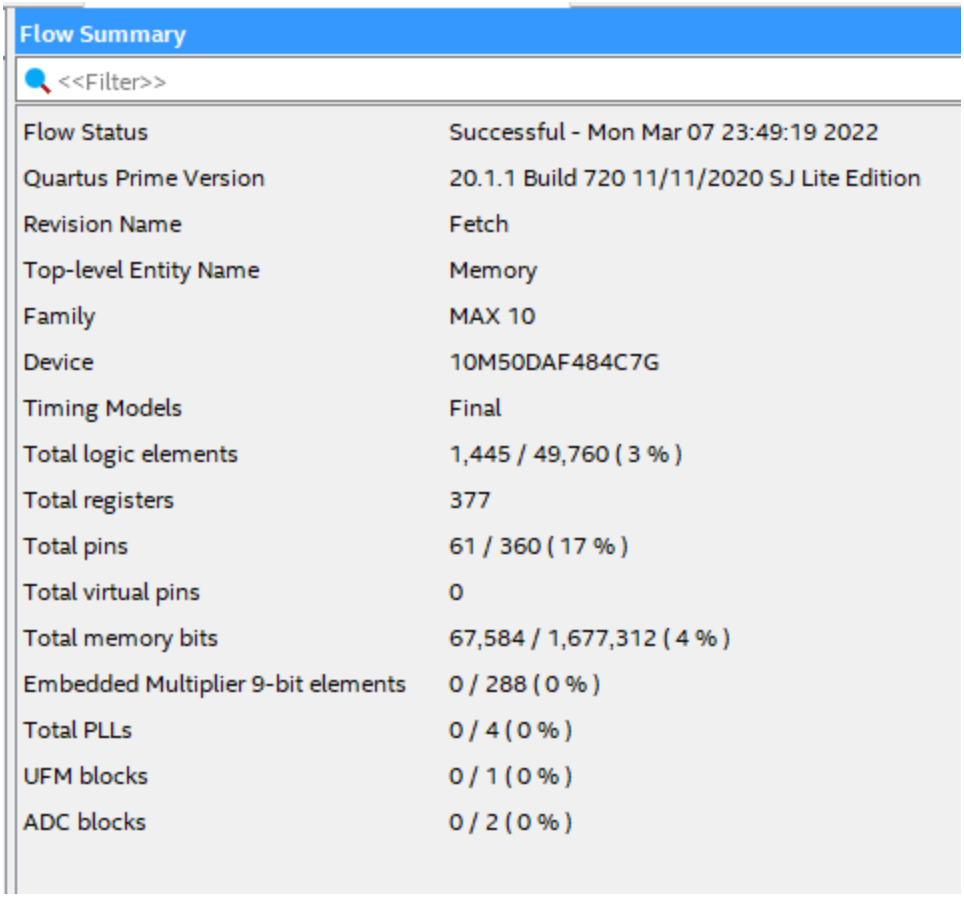


Figure 3. ModelSim Waveform of 6 Factorial With No NOPs

Where implementing data and control hazards takes a hit is in resource utilization. Comparing our implementation results that are summarized in Figure 4 to the results from lab 6, shows that the data/control hazards added 357 LUTs and 222 more registers. The increase in LUTs is completely within reason but the number of registers more than doubled. This was surprising to us as we were thinking there would maybe be a 50% increase. But this tradeoff is well worth it considering the over 2x performance increase.



Flow Summary

<<Filter>>

Flow Status	Successful - Mon Mar 07 23:49:19 2022
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	Fetch
Top-level Entity Name	Memory
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	1,445 / 49,760 (3 %)
Total registers	377
Total pins	61 / 360 (17 %)
Total virtual pins	0
Total memory bits	67,584 / 1,677,312 (4 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 4. Implementation Results

By far the hardest part was getting the jump and branch instructions to stall or invalidate the correct number of instructions. The biggest impedance was getting the PC counter to be minus one when stalling or jumping so that the correct instruction was the next one out once the jump was complete. Most of this frustration was caused by the ROM IP block as it forces the inputs to be registered, meaning that we had to stall and subtract one from the PC count a clock cycle earlier than we originally thought.

We also tested our factorial example by switching the RS1 and RS2 registers for ADD instructions to make sure that things were still working. To our surprise we found that the ADD instruction after the LW instruction failed to handle the data hazard correctly with the swapped registers. We were able to add more control logic to account for this, and we got it working again.

Conclusion

Overall, we are very pleased with the outcome of this lab. It was thrilling to see the working processor after spending considerable amounts of time on it. We have an

interest in making our processor work for both types of control hazards so that eventually we could have it perform branch prediction. For now we are satisfied with just handling the one case of branch predictions. We successfully handled all data hazards as well, and we were able to run the factorial program on our DLX processor without any NOPs.

Appendix A

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

entity DLX_Execute is
    port
    (
        clk             : in std_logic;
        id_ex_invalid   : in std_logic;
        opcode          : in std_logic_vector(c_DLX_OPCODE_WIDTH-1 downto 0);
        id_ex_rd_en     : in std_logic;
        id_ex_rd        : in std_logic_vector(c_DLX_REG_ADDR_WIDTH-1 downto 0);

        -- data hazards
        id_ex_rs1       : in std_logic_vector(c_DLX_REG_ADDR_WIDTH-1 downto 0);
        id_ex_rs2       : in std_logic_vector(c_DLX_REG_ADDR_WIDTH-1 downto 0);

        mem_wb_rd       : in std_logic_vector(c_DLX_REG_ADDR_WIDTH-1 downto 0);
        rd_mem_data     : in std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);

        -- ALU operand 0
        -- sel_pc       : in std_logic;
        pc_counter      : in std_logic_vector(c_DLX_PC_WIDTH-1 downto 0);
        operand_0       : in std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);

        -- ALU operand 1
        sel_immediate   : in std_logic;
        immediate       : in std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
        operand_1       : in std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
        -- Outputs
        ex_mem_invalid  : out std_logic;
        ex_mem_opcode   : out std_logic_vector(c_DLX_OPCODE_WIDTH-1 downto 0);
        stall           : out std_logic;
        sel_mem_alu     : out std_logic;
        ex_mem_rd       : out std_logic_vector(c_DLX_REG_ADDR_WIDTH-1 downto 0);
        ex_mem_rd_en    : out std_logic;
        mem_wr_en       : out std_logic;
        mem_data        : out std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
        branch_taken    : out std_logic;
        sel_jump_link   : out std_logic;
        pc_counter_out  : out std_logic_vector(c_DLX_PC_WIDTH-1 downto 0);
        data_out        : out std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0)
    );

end entity;

architecture rtl of DLX_Execute is
    signal alu_out  : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    signal alu_in_0 : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    signal alu_in_1 : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    signal is_zero : std_logic;
    signal br_taken : std_logic;
    signal br : std_logic;
    signal br_taken_0 : std_logic;
    signal br_taken_1 : std_logic;
    signal mem_en : std_logic;
    signal mem_sel : std_logic;
    signal link_sel : std_logic;
    signal stalling : std_logic;
    signal reg_to_reg_alu : std_logic;
    signal data_hazard_0 : std_logic;
    signal data_hazard_1 : std_logic;
    signal data_hazard_0_0 : std_logic;
    signal data_hazard_1_1 : std_logic;
    signal data_hazard_0_0_0 : std_logic;
    signal data_hazard_1_1_1 : std_logic;
    signal alu_piped_data : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);

begin

    is_zero <= '1' when (((alu_in_0 = x"00000000") and (opcode = c_DLX_BEQZ)) or
```

```
                        ((alu_in_0 /= x"00000000") and (opcode = c_DLX_BNEZ)) or
                        (opcode >= c_DLX_J)) and (id_ex_invalid = '0' and br_taken = '0') else '0';

    reg_to_reg_alu <= '1' when sel_immediate = '0' and opcode >= c_DLX_ADD and opcode <= c_DLX_SNEI
else '0';
    data_hazard_0 <= '1' when id_ex_rs1 = ex_mem_rd and opcode /= "000000" else '0';
    data_hazard_1 <= '1' when id_ex_rs2 = ex_mem_rd and reg_to_reg_alu = '1' else '0';
    data_hazard_0_0 <= '1' when id_ex_rs1 = mem_wb_rd and opcode /= "000000" else '0';
    data_hazard_1_1 <= '1' when id_ex_rs2 = mem_wb_rd and reg_to_reg_alu = '1' else '0';
    data_hazard_0_0_0 <= '1' when id_ex_rs1 = ex_mem_rd and ex_mem_opcode /= "000000"else '0';
    data_hazard_1_1_1 <= '1' when id_ex_rs2 = ex_mem_rd and ex_mem_opcode >= c_DLX_ADD and
ex_mem_opcode <= c_DLX_SNEI else '0';
    stall <= '1' when (data_hazard_0_0_0 = '1' or data_hazard_1_1_1 = '1') and stalling = '1' else '0';

    mem_en <= '1' when opcode = c_DLX_SW else '0';
    mem_sel <= '1' when opcode = c_DLX_LW else '0';
    link_sel <= '1' when opcode = c_DLX_JAL or opcode = c_DLX_JALR else '0';
    data_out <= alu_piped_data;

    p_UPPER_ALU_MUX : process(all)
    begin
        if data_hazard_0 = '0' and data_hazard_0_0 = '0' then
            alu_in_0 <= operand_0;
        elsif data_hazard_0_0 = '0' then
            alu_in_0 <= alu_piped_data;
        else
            alu_in_0 <= rd_mem_data;
        end if;
    end process;

    p_LOWER_ALU_MUX : process(all)
    begin
        if sel_immediate = '1' then
            alu_in_1 <= immediate;
        elsif data_hazard_1 = '0' and data_hazard_1_1 = '0' then
            alu_in_1 <= operand_1;
        elsif data_hazard_1_1 = '0' then
            alu_in_1 <= alu_piped_data;
        else
            alu_in_1 <= rd_mem_data;
        end if;
    end process;

    p_PIPELINE_REGISTER : process(clk)
        begin
            if rising_edge(clk) then
                if stall = '1' then
                    branch_taken <= branch_taken;
                    alu_piped_data <= alu_piped_data;
                    mem_wr_en <= mem_wr_en;
                    mem_data <= mem_data;
                    ex_mem_rd_en <= ex_mem_rd_en;
                    ex_mem_rd <= ex_mem_rd;
                    sel_mem_alu <= sel_mem_alu;
                    sel_jump_link <= sel_jump_link;
                    pc_counter_out <= pc_counter_out;
                    ex_mem_opcode <= ex_mem_opcode;
                    stalling <= '0';
                else
                    stalling <= mem_sel;
                    branch_taken <= is_zero;
                    alu_piped_data <= alu_out;
                    mem_wr_en <= mem_en;
                    mem_data <= operand_1;
                    ex_mem_rd_en <= id_ex_rd_en;
                    ex_mem_rd <= id_ex_rd;
                    sel_mem_alu <= mem_sel;
                    sel_jump_link <= link_sel;
                    pc_counter_out <= pc_counter;
                    ex_mem_opcode <= opcode;
                end if;
            end if;
        end process;

        br <= '1' when ex_mem_opcode = c_DLX_BEQZ or ex_mem_opcode = c_DLX_BNEZ else '0';
        br_taken <= br_taken_0 or br_taken_1 or (branch_taken and br);
```

```
        p_JUMP_STALL : process(clk)
        begin
            if rising_edge(clk) then
                ex_mem_invalid <= id_ex_invalid or br_taken;
                br_taken_0 <=  branch_taken and br;
                br_taken_1 <= br_taken_0;
            end if;
        end process;

    REG: entity work.DLX_ALU(rtl)
        port map (
            opcode => opcode,
            operand_0 => alu_in_0,
            operand_1 => alu_in_1,
            alu_out => alu_out
        );

end rtl;
```