Lab 5 Report

Matthew Crump A01841001
Nicholas Williams A02057223

Objectives

Build a functional DLX execute stage to be used in later labs. Learn how to build an ALU with all of its operations and find ways to make the jump and link instructions work. Simulate the execute stage with the fetch stage and decode stage from the previous labs.

Procedure

After reviewing the execute lecture material, we decided to put the ALU in it's own VHDL file in case we ever auto generate it and to make the main execute stage file less cluttered. All of the modules/components in the execute stage were combinational circuits and fed into the EX/MEM pipeline registers as shown in Figure 1. From the previous lab, we created two signals to control the muxes that fed the ALU. A select line for the upper mux that switches between the PC and operand 0. Another, for the lower mux that switches between the immediate value and operand 1. We didn't want to decode the opcode to figure out what the muxes should be as we thought that would add more combinational delay which could affect timing on the ALU. This way the ALU had the full clock-cycle to calculate its output.

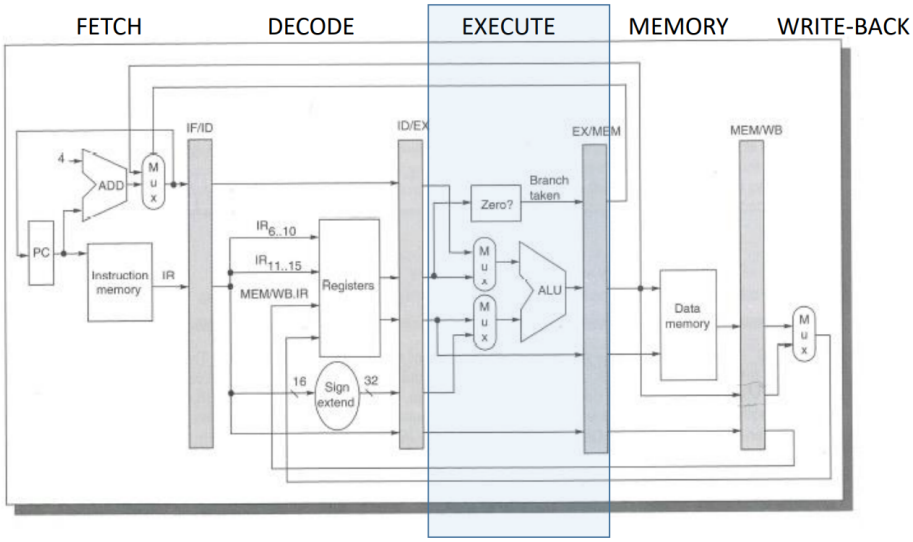

Figure 1. DLX Pipeline Diagram

The execute stage took in the opcode from the decode stage for use in the ALU. However, the opcode was not forwarded to the next stage of the pipeline. From here, control lines would leave the execute stage as it was less bits. The only control signals needed were the write enable for the data memory, and a write enable for write back to the registers. The write back address was passed on to the next stage unaltered. For the zero detect, we checked operand 0's value and ANDed that with the branch instruction. This was also ORed with the jump instructions as the PC will always take on a new value regardless of what operand 0 is.

To accommodate the requirement that R0 must always be zero, we decided to make it be enforced by the compiler. This way extra logic would not be needed and this would allow the user to get feedback before the program ran. If they accidentally wrote to R0 it would be harder to troubleshoot where the program went wrong or why it did not yield the correct result. The compiler can give them instant feedback and is much easier to correct before the program is uploaded to the CPU. Finally, to make things simpler while decoding and executing jump instructions, we made the instruction itself (in the rd section) contain the R31 address. This limits the range that jump addresses can go to but makes it equal with the branch instructions.

When we finished writing the modules for the execute stage we wrote a testbench for simulating the modules in ModelSim. We started with the testbench that we had created to simulate the fetch and decode stages of the pipeline. We tested the factorial example on our pipeline and specifically made sure that the output of the ALU was correct after each instruction. We also made sure that the zero block was outputting correctly. We then wrote another assembly file to test against and compiled it. The assembly file contained shift instructions and bitwise operations to be performed since those instructions were not well represented in the other example assembly files we had.

Results

Even though this design at this stage will not be placed on the board, we created a wrapper for the 3 completed stages. This revealed that certain things that we did in the ALU would not compile as written. For example:

alu_out <= x"00000001" when operand_0 /= x"00000000" else x"00000000"

These signals could not be assigned directly to the output of the ALU but had to have another signal created to do that operation and that was then assigned to the ALU output. The shift operations also needed slightly different unsigned and integer casting.

We used the decode stage's testbench and were able to remove a lot of the stimulus signals as the execute stage now took those over. However, it was now extremely confusing to look at the simulation results. There are almost too many signals to keep track of. We quickly realized that our program was not doing quite what we expected. This was because the results were not saved to the registers until several clock cycles later. Normally, there would be pipeline stalls but that logic does not exist yet. To fix this, we added 4 NOPs in between each instruction. This made it much easier to follow each instruction through each stage of the pipeline, and it fixed the result not being in the register by the time the next instruction was executed.

In the first run of simulations, we were not getting the correct output. The factorial program would stay in the first function call forever and keep adding up the factorial sum. This was caused by the jump and link instructions not working correctly. The CPU would jump to the correct location but the jump location was also saved to the R31 register instead of the address to where the PC was before the jump. This caused the function call to never exit. This was corrected by removing the PC/operand 0 mux and making the PC go to the pipeline registers separate from the ALU output. From here it would enter a simulated mux in the writeback stage that would select between the ALU output, data memory, or PC. Running the simulation again, the correct addresses were saved into the R31 register.

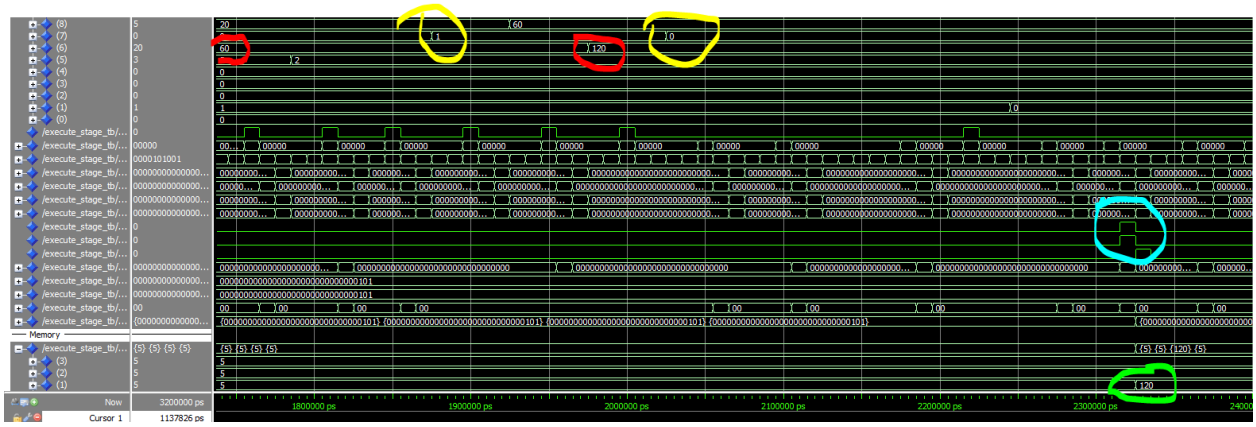

Figure 2. ModelSim waveform

Figure 2 shows the waveform for the simulated factorial example. We have circled the factorial count accumulating in register 6 in red. After the loop iteration counter becomes zero (circled in yellow), the memory write enable goes high (circled in blue). The final factorial result for 5 factorial which is 120 is then stored in the data memory (circled in green).

Figure 3 shows the waveform for the simulation of shift instructions and bitwise operations. Registers 1 and 2 were used for all of the instructions except the immediate instructions. For immediate instructions, register 1 and the value 2 were used. Registers 3-8 hold the output for the instructions SLL, SLLI, SRL, SRLI, SRA, and SRAI respectively. Registers 9-14 hold the output for the instructions AND, ANDI, OR, ORI, XOR, and XORI respectively. All of these operations seemed to perform correctly for this instruction set. We did in the meantime find an error in our compiler though when we realized that the data values were limited to the maximum value that a 16 bit register can hold and not a 32 bit register. Immediate instructions can only handle 16 bits for the immediate values, but the data part of the assembly code can receive 32 bit values.

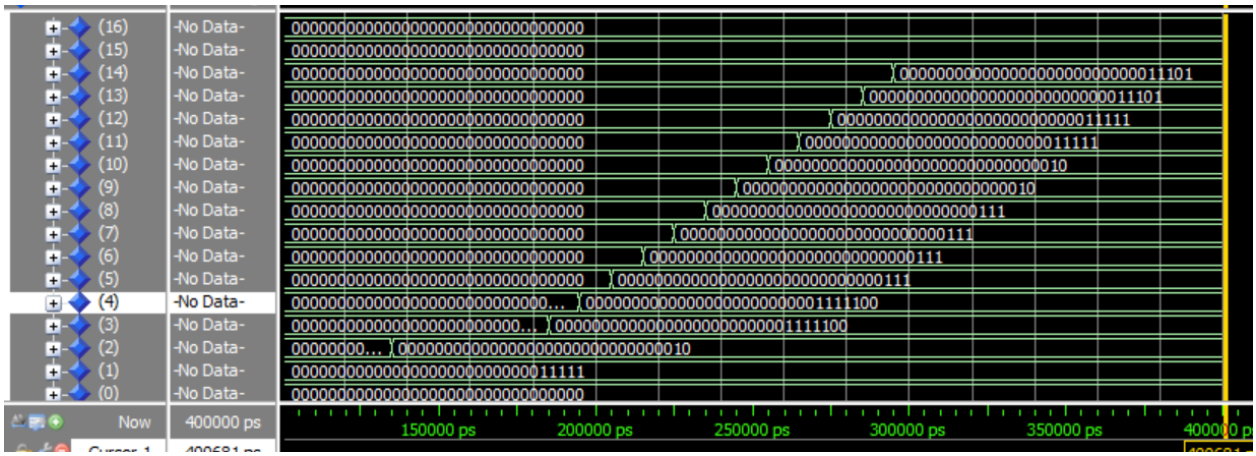

Figure 3. ModelSim waveform for shifts and bitwise operations

Conclusion

Overall, we successfully built an ALU inside of our execute stage and successfully demonstrated that our pipeline obtained the result for 5 factorial. We then demonstrated that our execute stage handles the other instructions correctly. We realized along the way that we need to implement stage stalls at some point. We are starting to realize the amount of testing that needs to be done in order to iron out some of the corner cases involved in this pipeline. We are pleased with the progress of our pipeline and look forward to improving it more.

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;
        opcode          : in std_logic_vector(c_DLX_OPCODE_WIDTH-1 downto 0);
        wr_en           : in std_logic;
        wr_addr         : in std_logic_vector(c_DLX_REG_ADDR_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
        sel_mem_alu     : out std_logic;
        mem_wr_en       : out std_logic;
        mem_data        : out std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
        wr_back_en      : out std_logic;
        wr_back_addr    : out std_logic_vector(c_DLX_REG_ADDR_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 mem_en : std_logic;
    signal mem_sel : std_logic;
    signal link_sel : std_logic;
begin

    is_zero <= '1' when ((operand_0 = x"00000000") and (opcode = c_DLX_BEQZ)) or
                        ((operand_0 /= x"00000000") and (opcode = c_DLX_BNEZ)) or
                        (opcode >= c_DLX_J) else '0';
    --alu_in_0 <= pc_counter when sel_pc = '1' else operand_0;
    alu_in_1 <= immediate when sel_immediate = '1' else operand_1;
    mem_en <= '1' when opcode = c_DLX_SW else '0';
    mem_sel <= '1' when opcode = c_DLX_SW or opcode = c_DLX_LW else '0';
    link_sel <= '1' when opcode = c_DLX_JAL or opcode = c_DLX_JALR else '0';

    p_PIPELINE_REGISTER : process(clk)
        begin
            if rising_edge(clk) then
                branch_taken <= is_zero;
                data_out <= alu_out;
                mem_wr_en <= mem_en;
                mem_data <= operand_1;
                wr_back_en <= wr_en;
                wr_back_addr <= wr_addr;
                sel_mem_alu <= mem_sel;
                sel_jump_link <= link_sel;
```

```
                pc_counter_out <= pc_counter;
            end if;
        end process;

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

end rtl;
```

Appendix B

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

entity DLX_ALU is
    port
    (
        opcode      : in std_logic_vector(c_DLX_OPCODE_WIDTH-1 downto 0);
        operand_0   : in std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
        operand_1   : in std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
        alu_out     : out std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0)
    );
end entity;

architecture rtl of DLX_ALU is
    signal slt : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    signal sltu : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    signal sgt : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    signal sgtu : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    signal sle : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    signal sleu : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    signal sge : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    signal sgeu : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    signal seq : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    signal sne : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);

begin

    slt <= x"00000001" when signed(operand_0) < signed(operand_1) else x"00000000";
    sltu <= x"00000001" when unsigned(operand_0) < unsigned(operand_1) else x"00000000";
    sgt <= x"00000001" when signed(operand_0) > signed(operand_1) else x"00000000";
    sgtu <= x"00000001" when unsigned(operand_0) > unsigned(operand_1) else x"00000000";
    sle <= x"00000001" when signed(operand_0) <= signed(operand_1) else x"00000000";
    sleu <= x"00000001" when unsigned(operand_0) <= unsigned(operand_1) else x"00000000";
    sge <= x"00000001" when signed(operand_0) >= signed(operand_1) else x"00000000";
    sgeu <= x"00000001" when unsigned(operand_0) >= unsigned(operand_1) else x"00000000";
    seq <= x"00000001" when operand_0 = operand_1 else x"00000000";
    sne <= x"00000001" when operand_0 /= operand_1 else x"00000000";

    process(opcode, operand_0, operand_1, slt, sltu, sgt, sgtu, sle, sleu, sge, sgeu, seq,
sne)
    begin
        case opcode is
            when c_DLX_NOP   =>
                alu_out <= x"00000000";
            when c_DLX_LW    =>
                alu_out <= operand_0 + operand_1;
            when c_DLX_SW    =>
                alu_out <= operand_0 + operand_1;
            when c_DLX_ADD   =>
                alu_out <= operand_0 + operand_1;
            when c_DLX_ADDI  =>
                alu_out <= operand_0 + operand_1;
            when c_DLX_ADDU  =>
                alu_out <= operand_0 + operand_1;
            when c_DLX_ADDUI =>
                alu_out <= operand_0 + operand_1;
            when c_DLX_SUB   =>
                alu_out <= operand_0 - operand_1;
            when c_DLX_SUBI  =>
                alu_out <= operand_0 - operand_1;
            when c_DLX_SUBU  =>
```

```
                alu_out <= operand_0 - operand_1;
            when c_DLX_SUBUI =>
                alu_out <= operand_0 - operand_1;
            when c_DLX_AND   =>
                alu_out <= operand_0 and operand_1;
            when c_DLX_ANDI  =>
                alu_out <= operand_0 and operand_1;
            when c_DLX_OR    =>
                alu_out <= operand_0 or operand_1;
            when c_DLX_ORI   =>
                alu_out <= operand_0 or operand_1;
            when c_DLX_XOR   =>
                alu_out <= operand_0 xor operand_1;
            when c_DLX_XORI  =>
                alu_out <= operand_0 xor operand_1;
            when c_DLX_SLL   =>
                alu_out <= std_logic_vector(shift_left(unsigned(operand_0),
to_integer(unsigned(operand_1))));
            when c_DLX_SLLI  =>
                alu_out <= std_logic_vector(shift_left(unsigned(operand_0),
to_integer(unsigned(operand_1))));
            when c_DLX_SRL   =>
                alu_out <= std_logic_vector(shift_right(unsigned(operand_0),
to_integer(unsigned(operand_1))));
            when c_DLX_SRLI  =>
                alu_out <= std_logic_vector(shift_right(unsigned(operand_0),
to_integer(unsigned(operand_1))));
            when c_DLX_SRA   =>
                alu_out <= std_logic_vector(shift_right(signed(operand_0),
to_integer(unsigned(operand_1))));
            when c_DLX_SRAI  =>
                alu_out <= std_logic_vector(shift_right(signed(operand_0),
to_integer(unsigned(operand_1))));
            when c_DLX_SLT   =>
                alu_out <= slt;
            when c_DLX_SLTI  =>
                alu_out <= slt;
            when c_DLX_SLTU  =>
                alu_out <= sltu;
            when c_DLX_SLTUI =>
                alu_out <=  sltu;
            when c_DLX_SGT   =>
                alu_out <= sgt;
            when c_DLX_SGTI  =>
                alu_out <= sgt;
            when c_DLX_SGTU  =>
                alu_out <=  sgtu;
            when c_DLX_SGTUI =>
                alu_out <=  sgtu;
            when c_DLX_SLE   =>
                alu_out <= sle;
            when c_DLX_SLEI  =>
                alu_out <= sle;
            when c_DLX_SLEU  =>
                alu_out <=  sleu;
            when c_DLX_SLEUI =>
                alu_out <=  sleu;
            when c_DLX_SGE   =>
                alu_out <= sge;
            when c_DLX_SGEI  =>
                alu_out <= sge;
            when c_DLX_SGEU  =>
                alu_out <=  sgeu;
            when c_DLX_SGEUI =>
                alu_out <= sgeu;
            when c_DLX_SEQ   =>
                alu_out <= seq;
```

```
            when c_DLX_SEQI  =>
                alu_out <= seq;
            when c_DLX_SNE   =>
                alu_out <= sne;
            when c_DLX_SNEI  =>
                alu_out <= sne;
            when c_DLX_BEQZ  =>
                alu_out <= operand_1;
            when c_DLX_BNEZ  =>
                alu_out <= operand_1;
            when c_DLX_J     =>
                alu_out <= operand_1;
            when c_DLX_JR    =>
                alu_out <= operand_1;
            when c_DLX_JAL   =>
                alu_out <= operand_1;
            when c_DLX_JALR  =>
                alu_out <= operand_1;
            when others =>
                alu_out <= x"00000000";

        end case;
    end process;

end rtl;
```