Lab 4 Report

Matthew Crump A01841001
Nicholas Williams A02057223

Objectives

Build a functional DLX decode stage to be used in later labs. Learn how to initiate a dual-port RAM module by inferring one in VHDL. Simulate the decode stage with the fetch stage from the previous lab.

Procedure

Following what we learned in class, we laid out what inputs and outputs were needed by the decode stage. The inputs are the instruction from the fetch stage, the write data, the write address, and a write enable bit. For the outputs, we had the two registers, the immediate value extended, and also the PC output, as we did not want to have a pipeline process in our top module. There is also the instruction as an output, but this is subject to change and may switch to being control signals depending on how the execute stage functions.

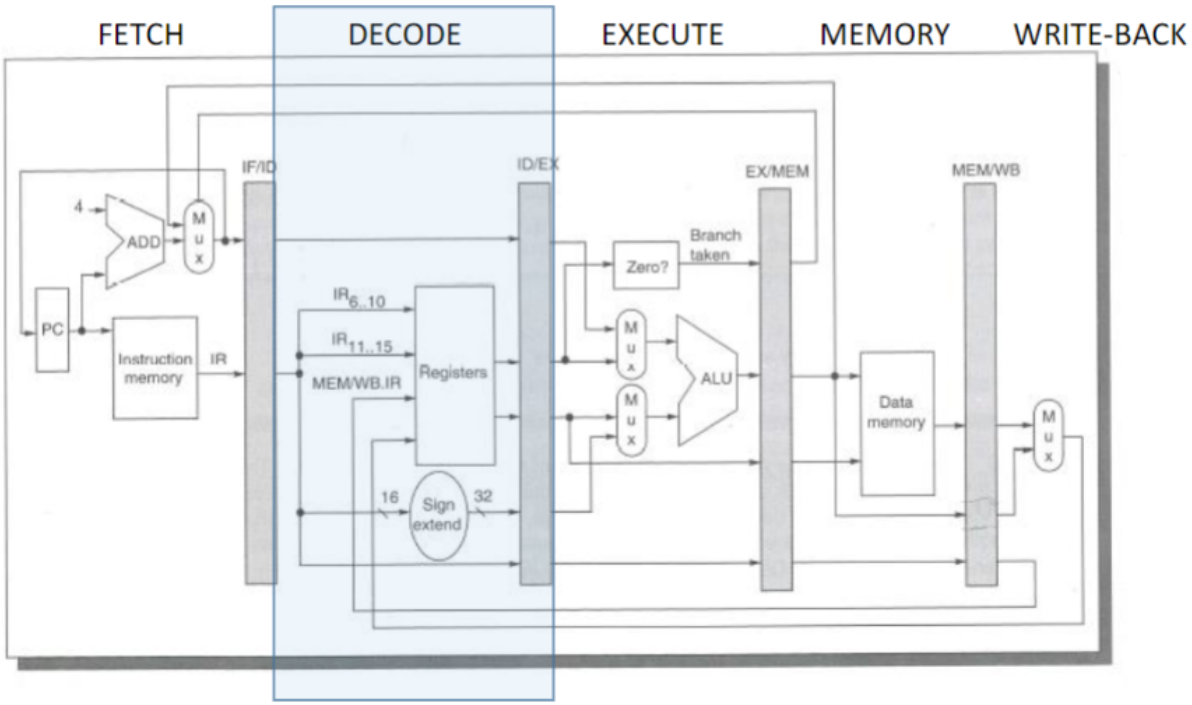

Figure 1. DLX Pipeline Diagram

The thirty-two 32-bit registers were instantiated as RAM, by inferring a dual-port RAM module with a depth and width of 32. The initial code for this was taken from the examples shown in class and then modified to meet the needs of the decode stage. Sign extension also took place at this stage for the immediate value. The sign was extended if the MSB of the immediate value was 1 and the instruction was for a signed operation. If the instruction was an unsigned operation then 0s were appended onto the front of the bits.

We also revisited the DLX Assembly lab to add a few additional features. We had our assembler take in an Excel spreadsheet that has a list of all instructions, op-codes, and the type of operation. This was used to then generate a Python file that contained lists and dictionaries to be used by the main Python assembly file. This same program also generated a VHDL package file (see Appendix C). This was done so that if we made changes to the opcodes or added more opcodes we could easily generate the new files or code very quickly with minimal effort.

We then wrote a testbench to simulate our design. The testbench built off of the previous lab by using the output of the fetch stage to drive the inputs of the decode stage. The testbench used for simulating the fetch stage handled the branch taken and branch address inputs already. For this testbench, we kept the process to simulate the fetch stage and then added another process to test the inputs of the decode stage. We had to manually manipulate the write enable, write address, and write data lines since the future stages that will handle those lines are not implemented yet.

Results

We first wrote the code that handled all of the inputs and outputs to the decode stage that appear in the system diagram that we have seen in class. We then realized that the muxes in the execute stage will need to know which line to select, and the immediate block in the decode stage needs to know if it received a signed operation or not. We created a program counter select line and an immediate value select line to send to the execute state of the pipeline.

Instead of using a case statement like we initially planned we realized there was a pattern in the opcode design and decided to use continuous logic to see if the instruction was a signed or an immediate value. We realized that all of the immediate values in the list were even. There were only 12 signed operations so we just hard coded them in. Once we were done writing the code for our decode stage we tried

compiling. After fixing a few syntax errors we finally got our project to compile successfully.

We then worked on putting together a simulation to test the fetch and decode stages of the pipeline simultaneously. We began by copying our testbench that we used to simulate the fetch stage. We added a port mapping for the decode stage, and we added signals to connect the fetch stage to the decode stage. We already had a process from the last testbench available that stimulated the branching and jumping of the fetch stage. We then wrote a process to control which values got written to which registers. We decided to just write n to every register where n increased every clock cycle.

We then compiled our simulation and simulated it. Everything seemed to be working except the output of the registers block remained uninitialized. This was quickly fixed by initializing the 2D array to all zeros for simulation. We then looked over all of the signals to find potential errors. We realized that register 0 was being written to. We looked at our test bench and found that our logic for deciding when to write to the registers based on the opcodes was incorrect. We were using an or operation instead of an and operation. We could not find another error in the signals after fixing that mistake.

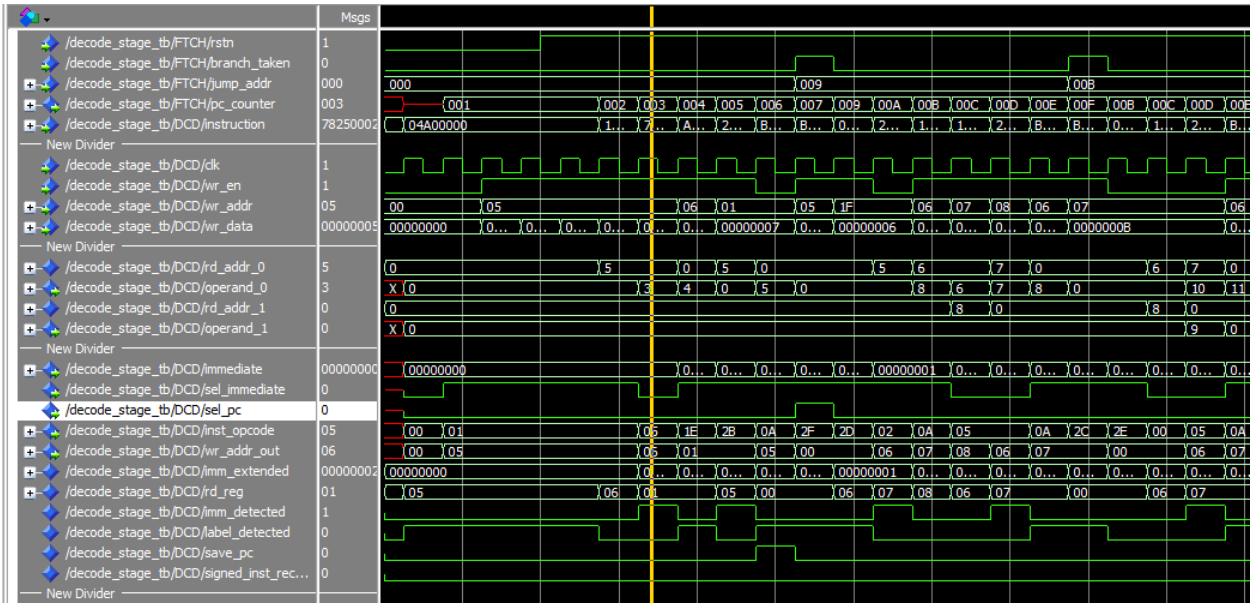

Figure 2. Modelsim Waveform

Figure 2 shows the majority of the signals contained in our simulated waveform. We were able to see that reading from registers we were obtaining the true value from inside the register. Figure 3 shows the 32 registers and the values stored inside of them. We can see that as an instruction comes the true registers are read or the true

immediate value is sign extended. We can also see our control signals be generated that will be used in the execute stage.

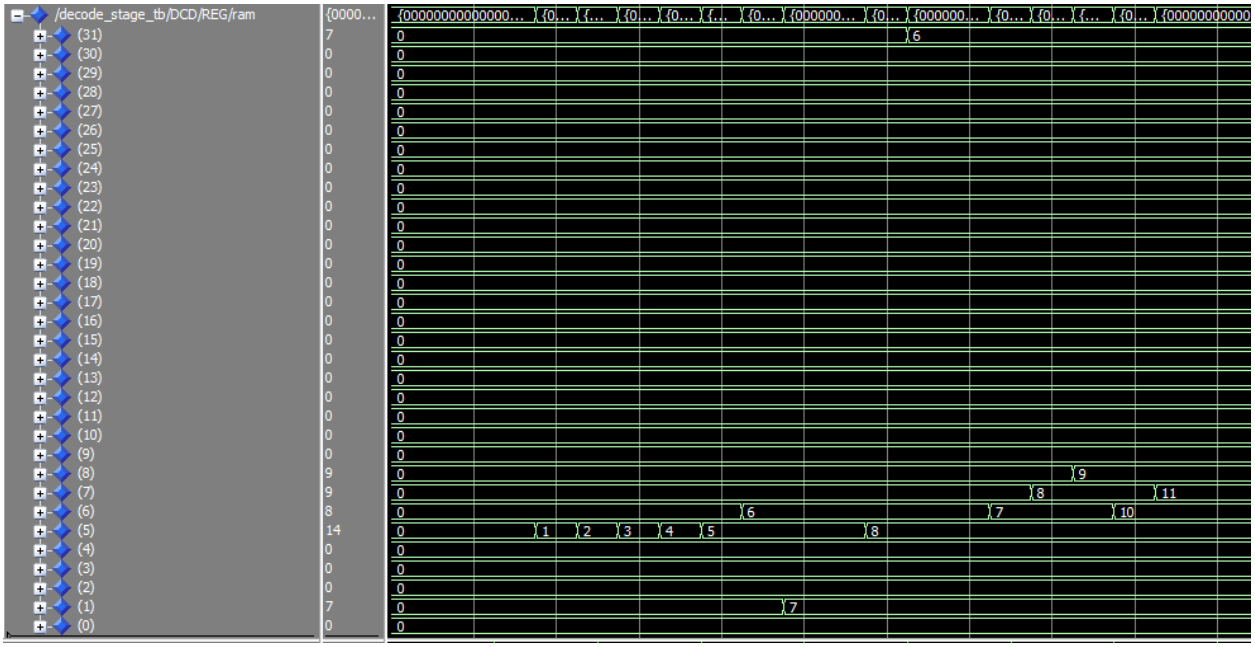

Figure 3. ModelSim Waveform of 32 registers

Conclusion

Overall, we successfully designed a decode stage and simulated it working alongside the fetch stage. We learned how to create a VHDL package file. We learned how to create a RAM block of thirty-two 32-bit registers using inference. We also gained more experience using ModelSim. We look forward to implementing the execute stage, and we believe we have a good trajectory to design the rest of the stages of the pipeline.

Appendix A

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

entity DLX_Decode is
    port
    (
        clk         : in std_logic;
        instruction : in std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
        wr_en       : in std_logic;
        wr_addr     : in std_logic_vector(c_DLX_REG_ADDR_WIDTH-1 downto 0);
        wr_data     : in std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
        pc_counter  : in std_logic_vector(c_DLX_PC_WIDTH-1 downto 0);
        operand_0   : out std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
        operand_1   : out std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
        immediate   : out std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
        sel_immediate : out std_logic;
        sel_pc      : out std_logic;
        inst_opcode : out std_logic_vector(c_DLX_OPCODE_WIDTH-1 downto 0);
        wr_addr_out : out std_logic_vector(c_DLX_REG_ADDR_WIDTH-1 downto 0);
        pc_counter_padded : out std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0)
    );

end entity;

architecture rtl of DLX_Decode is
    signal opcode : std_logic_vector(c_DLX_OPCODE_WIDTH-1 downto 0);
    signal imm : std_logic_vector(c_DLX_IMM_WIDTH-1 downto 0);
    signal imm_extended : std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    signal rd_addr_0 : std_logic_vector(c_DLX_REG_ADDR_WIDTH-1 downto 0);
    signal rd_addr_1 : std_logic_vector(c_DLX_REG_ADDR_WIDTH-1 downto 0);
    signal rd_reg : std_logic_vector(c_DLX_REG_ADDR_WIDTH-1 downto 0);
    signal imm_detected : std_logic;
    signal label_detected : std_logic;
    signal save_pc : std_logic;
    signal signed_inst_received : std_logic;

begin
    save_pc <= '1' when (opcode >= c_DLX_JAL) else '0';

    label_detected <= '1' when ((opcode >= c_DLX_LW and opcode <= c_DLX_SW) or
(opcode >= c_DLX_BEQZ)) else '0';

    imm_detected <= '1' when ((opcode >= c_DLX_ADDI and opcode <= c_DLX_SNEI)
and (opcode(0) = '0')) else '0';

    signed_inst_received <= '1' when (opcode = c_DLX_ADD or opcode = c_DLX_ADDI
or opcode = c_DLX_SUB or opcode = c_DLX_SUBI or
                                                opcode = c_DLX_SLT or opcode =
c_DLX_SLTI or opcode = c_DLX_SGT or opcode = c_DLX_SGTI or
                                                opcode = c_DLX_SLE or opcode =
c_DLX_SLEI or opcode = c_DLX_SGE or opcode = c_DLX_SGEI) else '0';
```

```
    p_SIGN_EXTEND : process(imm)
    begin
        if (imm(c_DLX_IMM_WIDTH-1) = '1') and (signed_inst_received = '1') then
-- add signed instruction check
            imm_extended <= std_logic_vector(resize(signed(imm),
c_DLX_WORD_WIDTH));
        else
            imm_extended <= std_logic_vector(resize(unsigned(imm),
c_DLX_WORD_WIDTH));
        end if;
    end process;

    p_PIPELINE_REGISTER : process(clk)
    begin
        if rising_edge(clk) then
            immediate <= imm_extended;
            sel_immediate <= imm_detected or label_detected;
            sel_pc <= save_pc;
            inst_opcode <= opcode;
            pc_counter_padded <= std_logic_vector(resize(unsigned(pc_counter),
c_DLX_WORD_WIDTH));
            wr_addr_out <= rd_reg;
        end if;
    end process;

    opcode <= instruction(31 downto 31-c_DLX_OPCODE_WIDTH+1);
    rd_addr_0 <= instruction(20 downto 20-c_DLX_REG_ADDR_WIDTH+1);
    rd_addr_1 <= instruction(15 downto 15-c_DLX_REG_ADDR_WIDTH+1);
    imm <= instruction(15 downto 15-c_DLX_IMM_WIDTH+1);
    rd_reg <= instruction(25 downto 25-c_DLX_REG_ADDR_WIDTH+1);

    REG: entity work.DLX_Registers(rtl)
        port map (
            clk => clk,
            wr_en => wr_en,
            wr_addr => wr_addr,
            wr_data => wr_data,
            rd_addr_0 => rd_addr_0,
            rd_addr_1 => rd_addr_1,
            rd_data_0 => operand_0,
            rd_data_1 => operand_1
        );

end rtl;
```

Appendix B

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

entity DLX_Registers is
    port
    (
        clk         : in std_logic;
        wr_en       : in std_logic;
        wr_addr     : in std_logic_vector(c_DLX_REG_ADDR_WIDTH-1 downto 0);
        wr_data     : in std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
        rd_addr_0   : in std_logic_vector(c_DLX_REG_ADDR_WIDTH-1 downto 0);
        rd_addr_1   : in std_logic_vector(c_DLX_REG_ADDR_WIDTH-1 downto 0);
        rd_data_0   : out std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
        rd_data_1   : out std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0)
    );

end entity;

architecture rtl of DLX_Registers is

    -- Build a 2-D array type for the RAM
    subtype word_t is std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0);
    type memory_t is array(c_NUM_OF_REGISTERS-1 downto 0) of word_t;

    -- Declare the RAM signal.
    signal ram : memory_t;

begin

    process(clk)
    begin
        if(rising_edge(clk)) then
            if(wr_en = '1') then
                ram(to_integer(unsigned(wr_addr))) <= wr_data;
            end if;

            rd_data_0 <= ram(to_integer(unsigned(rd_addr_0)));
            rd_data_1 <= ram(to_integer(unsigned(rd_addr_1)));
        end if;

    end process;

end rtl;
```

Appendix C

```
library ieee;
use ieee.math_real.all;
use ieee.std_logic_1164.all;

package dlx_package is

    constant c_DLX_PC_WIDTH : integer := 10;
    constant c_DLX_WORD_WIDTH : integer := 32;
    constant c_NUM_OF_REGISTERS : integer := 32;
    constant c_DLX_REG_ADDR_WIDTH : integer :=
integer(ceil(log2(real(c_NUM_OF_REGISTERS))));
    constant c_DLX_IMM_WIDTH : integer := 16;
    constant c_DLX_OPCODE_WIDTH : integer := 6;

    -- Instructions

    constant c_DLX_NOP      : std_logic_vector(5 downto 0) := "000000";
    constant c_DLX_LW       : std_logic_vector(5 downto 0) := "000001";
    constant c_DLX_SW       : std_logic_vector(5 downto 0) := "000010";
    constant c_DLX_ADD      : std_logic_vector(5 downto 0) := "000011";
    constant c_DLX_ADDI     : std_logic_vector(5 downto 0) := "000100";
    constant c_DLX_ADDU     : std_logic_vector(5 downto 0) := "000101";
    constant c_DLX_ADDUI    : std_logic_vector(5 downto 0) := "000110";
    constant c_DLX_SUB      : std_logic_vector(5 downto 0) := "000111";
    constant c_DLX_SUBI     : std_logic_vector(5 downto 0) := "001000";
    constant c_DLX_SUBU     : std_logic_vector(5 downto 0) := "001001";
    constant c_DLX_SUBUI    : std_logic_vector(5 downto 0) := "001010";
    constant c_DLX_AND      : std_logic_vector(5 downto 0) := "001011";
    constant c_DLX_ANDI     : std_logic_vector(5 downto 0) := "001100";
    constant c_DLX_OR       : std_logic_vector(5 downto 0) := "001101";
    constant c_DLX_ORI      : std_logic_vector(5 downto 0) := "001110";
    constant c_DLX_XOR      : std_logic_vector(5 downto 0) := "001111";
    constant c_DLX_XORI     : std_logic_vector(5 downto 0) := "010000";
    constant c_DLX_SLL      : std_logic_vector(5 downto 0) := "010001";
    constant c_DLX_SLLI     : std_logic_vector(5 downto 0) := "010010";
    constant c_DLX_SRL      : std_logic_vector(5 downto 0) := "010011";
    constant c_DLX_SRLI     : std_logic_vector(5 downto 0) := "010100";
    constant c_DLX_SRA      : std_logic_vector(5 downto 0) := "010101";
    constant c_DLX_SRAI     : std_logic_vector(5 downto 0) := "010110";
    constant c_DLX_SLT      : std_logic_vector(5 downto 0) := "010111";
    constant c_DLX_SLTI     : std_logic_vector(5 downto 0) := "011000";
    constant c_DLX_SLTU     : std_logic_vector(5 downto 0) := "011001";
    constant c_DLX_SLTUI    : std_logic_vector(5 downto 0) := "011010";
    constant c_DLX_SGT      : std_logic_vector(5 downto 0) := "011011";
    constant c_DLX_SGTI     : std_logic_vector(5 downto 0) := "011100";
    constant c_DLX_SGTU     : std_logic_vector(5 downto 0) := "011101";
    constant c_DLX_SGTUI    : std_logic_vector(5 downto 0) := "011110";
    constant c_DLX_SLE      : std_logic_vector(5 downto 0) := "011111";
    constant c_DLX_SLEI     : std_logic_vector(5 downto 0) := "100000";
    constant c_DLX_SLEU     : std_logic_vector(5 downto 0) := "100001";
    constant c_DLX_SLEUI    : std_logic_vector(5 downto 0) := "100010";
    constant c_DLX_SGE      : std_logic_vector(5 downto 0) := "100011";
    constant c_DLX_SGEI     : std_logic_vector(5 downto 0) := "100100";
    constant c_DLX_SGEU     : std_logic_vector(5 downto 0) := "100101";
    constant c_DLX_SGEUI    : std_logic_vector(5 downto 0) := "100110";
    constant c_DLX_SEQ      : std_logic_vector(5 downto 0) := "100111";
    constant c_DLX_SEQI     : std_logic_vector(5 downto 0) := "101000";
```

```
    constant c_DLX_SNE      : std_logic_vector(5 downto 0) := "101001";
    constant c_DLX_SNEI     : std_logic_vector(5 downto 0) := "101010";
    constant c_DLX_BEQZ     : std_logic_vector(5 downto 0) := "101011";
    constant c_DLX_BNEZ     : std_logic_vector(5 downto 0) := "101100";
    constant c_DLX_J        : std_logic_vector(5 downto 0) := "101101";
    constant c_DLX_JR       : std_logic_vector(5 downto 0) := "101110";
    constant c_DLX_JAL      : std_logic_vector(5 downto 0) := "101111";
    constant c_DLX_JALR     : std_logic_vector(5 downto 0) := "110000";

end package dlx_package;

package body dlx_package is

end package body dlx_package;
```