Lab 3 Report

Matthew Crump A01841001
Nicholas Williams A02057223

Objectives

Build a functional DLX fetch stage to be used in later labs. Learn how to initiate a ROM IP module with a .mif file. Simulate the fetch stage with the IP ROM module.

Procedure

Following what we learned in class, we grouped the logical blocks of the fetch stage into two groups, combinational circuits and sequential circuits. The mux was the only combinational circuit used. The program counter (PC), ROM, and pipeline output for the mux were sequential circuits. The mux simply switched between the program counter plus one and the jump address. The mux output was connected to the pipeline register and the program counter. The last clocked process simply resets the PC or sets the PC to the output of the mux.

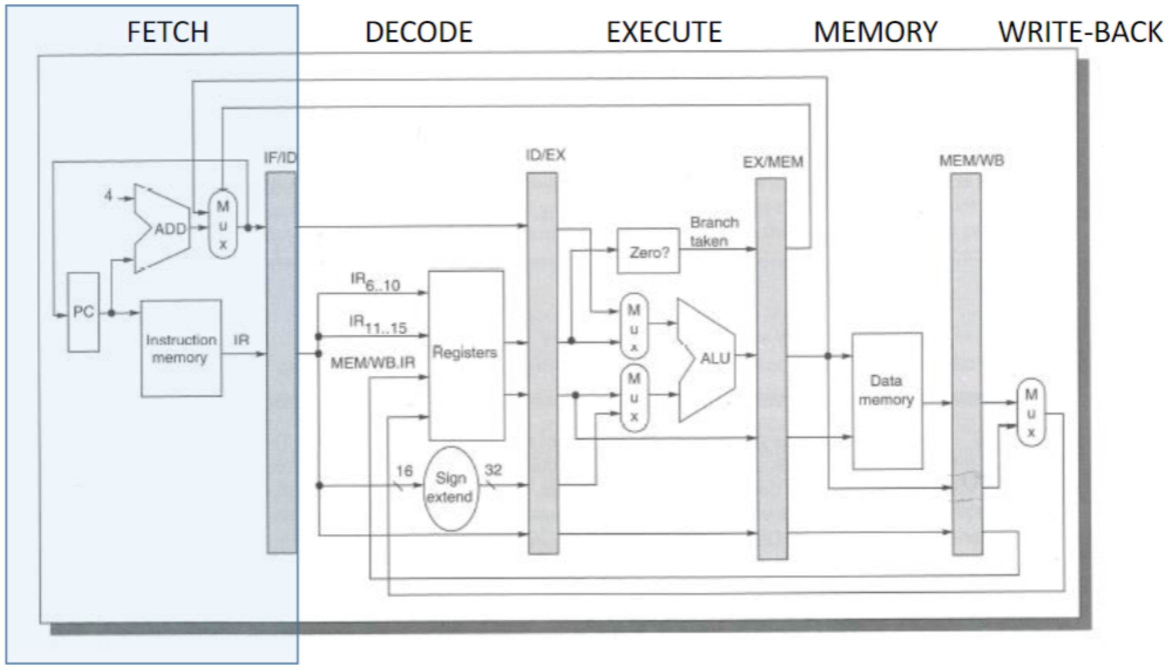

Figure 1. DLX Pipeline Diagram

The ROM was initialized with a .mif file that was generated from the factorial.dlx file. The presets of the ROM used all of the presets in the memory wizard except for the registered output. A simulation file was also generated so we could simulate our ROM in ModelSim. We then created a top file just to make sure that our DLX fetch stage would compile.

We then wrote a testbench to simulate our design. The testbench simply controls the clock signal, the branch_taken signal, and the jump_address signal. When a branch or jump instruction is detected our testbench sets the branch_taken signal high and then sets the jump_address to the appropriate address. For most jump and branch commands, the test bench simply set the jump_address to the 10 least significant bits in the instruction. For the jump register (JR) instruction our testbench was hardcoded to set the appropriate jump_address to 0x003 since that was the location in the .mif that used the jump and link (JAL) instruction.

Results

While compiling in Quartus, we encountered an error that the ROM module could not be synthesized, saying it needs to be switched to a “ERAM”. This error was caused because the Quartus settings did not allow for a bitstream with memory initialization. This setting was easily changed and allowed the design to be compiled without any errors.

From here, we created a testbench in ModelSim. Initially when we compiled and simulated our design in ModelSim we had an error that set the branch taken signal high for two clock cycles because there were two branch or jump instructions in a row in the .mif file. To make sure that our design could successfully perform the fetch stage for computing a factorial, we created signals that would not let a branch or jump instruction take place if a branch or jump instruction occurred the line prior. We then created counters for the amount of times that loops and functions could be called so that we could see every instruction take place and not get stuck in a loop.

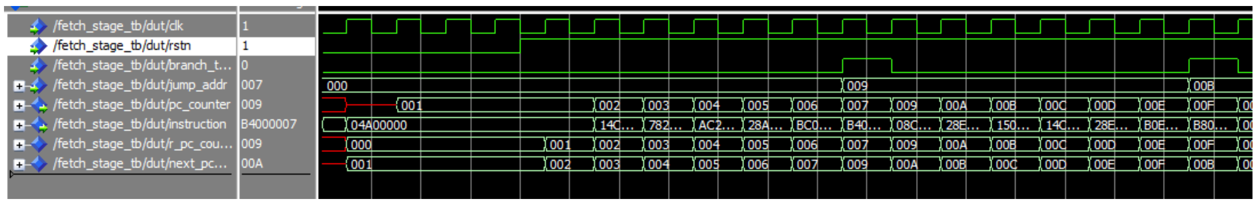
/fetch_stage_tb/dut/clk 1
/fetch_stage_tb/dut/rstn 1
/fetch_stage_tb/dut/branch_t... 0
/fetch_stage_tb/dut/jump_addr 007
/fetch_stage_tb/dut/pc_counter 009
/fetch_stage_tb/dut/instruction B4000007
/fetch_stage_tb/dut/r_pc_cou... 009
/fetch_stage_tb/dut/next_pc... 00A

Figure 2. Modelsim Waveform

Figure 2 shows a portion of the simulated waveform for our design. The program_counter signal shows the instruction address and then the actual instruction appears on the instruction signal the next clock cycle. The branch_taken signal goes high, and the jump_address is updated the clock cycle after the instruction signal receives a branch or jump instruction. The program_counter signal then receives the instruction address for the corresponding jump or branch, the clock cycle after the branch_taken signal goes high.

Conclusion

We successfully created the DLX fetch state of the DLX pipeline that will be used in future labs. We also created a testbench to simulate the design, and we are confident that it is working and ready to be implemented alongside the other stages of the pipeline. We learned how to instantiate a ROM IP module using a .mif file. We learned that future sections in the pipeline will need to ignore instructions that come directly after a branch or jump command until the jump address has successfully reached the program counter. Overall, we are ready and eager to design the next stage.

APPENDIX A

DLX Fetch VHDL Code

```vhdl
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_Fetch is
    port (
        clk : in std_logic;
        rstn : in std_logic;
        branch_taken : in std_logic;
        jump_addr : in std_logic_vector(c_DLX_PC_WIDTH-1 downto 0);
        pc_counter : out std_logic_vector(c_DLX_PC_WIDTH-1 downto 0);
        instruction : out std_logic_vector(c_DLX_WORD_WIDTH-1 downto 0)
    );
end DLX_Fetch;

architecture rtl of DLX_Fetch is
    signal r_pc_counter : std_logic_vector(c_DLX_PC_WIDTH-1 downto 0);
    signal next_pc_count : std_logic_vector(c_DLX_PC_WIDTH-1 downto 0);
begin

    p_2_TO_2_MUX : process (branch_taken, r_pc_counter, next_pc_count,
jump_addr)
    begin
        if branch_taken = '1' then
            next_pc_count <= jump_addr;
        else
            next_pc_count <= r_pc_counter + '1';
        end if;
    end process;

    p_PC_COUNTER : process(clk)
    begin
        if rising_edge(clk) then
            if rstn = '0' then
                r_pc_counter <= (others => '0');
            else
                r_pc_counter <= next_pc_count;
            end if;
        end if;
    end process;

    p_PIPELINE_REGISTER : process(clk)
    begin
        if rising_edge(clk) then
            pc_counter <= next_pc_count;
        end if;
    end process;

    ROM: entity work.Instruction_Memory(SYN)
        port map (
            address => r_pc_counter,
            clock => clk,
            q => instruction
        );

end rtl;
```