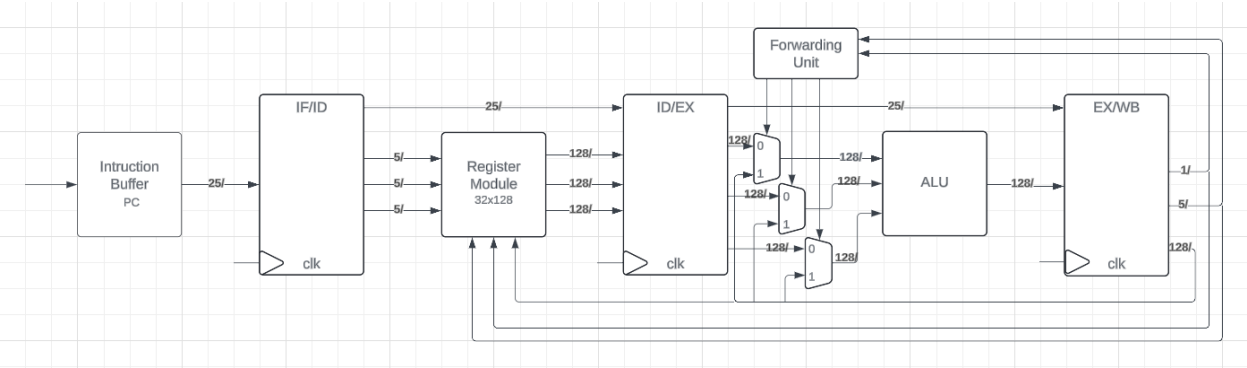
ESE 345 Final Project Report

By Kenneth Procacci and Matthew Huber

Goals:
The main goal of this project was to implement a 4-stage pipelined multimedia unit of a set of instructions using VHDL. This set of instructions are similar to those in Sony Cell SPU and Intel SSE architectures. The project served as a reminder and an opportunity for us to work with and improve our skills with VHDL. The project was designed in a way for us to work on smaller components pertaining to each of the stages and finally combine the components into the complete pipelined multimedia unit. The components included a multimedia alu, instruction buffers, register file, forwarding unit, and clock-edge sensitive pipeline buffers. When working on the project, we found that testing and simulating each component separately in Aldec Active-HDL before using it in the completed project made the debugging process much easier.

The purpose of using a pipelined design for a processor is to increase performance by allowing multiple computations to perform simultaneously. In our program for example, a single operation (one stage) can be performed in a single clock cycle (say 20ns for this example). Since there are four stages for an instruction to be fully read and executed, it will take 80ns for a single instruction to be executed. If there are multiple instructions and they are executed sequentially, the total execution time required is 80n ns for n instructions. Pipelining allows computations to run in parallel, since a different instruction can be executed in each of the stages simultaneously. As the first instruction is finished in stage 1 and moves on to stage 2, a new instruction can enter stage 1, and so on. As a result, the execution of a single instruction will still be 80ns, but each consecutive instruction will only take 20ns to complete following that first instruction. So the total execution time would be (80 + 20(n-1))ns or (20n + 60)ns.

Block Diagram:



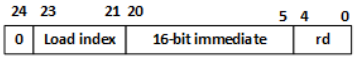
This diagram shows the 4-stage pipeline. Stage 1 is called the Instruction Fetch (IF) stage, and this is where the instruction buffer receives the machine instructions as input from a text file. The program counter is incremented upon every clock cycle, and its value determines which instruction to be output by the instruction buffer. Next, stage 2 is called the Instruction Decode (ID) stage. This is where the process retrieves the 25-bit instruction containing the addresses of the registers to be read and the register to write. It breaks up the instruction into its necessary components used in the ALU. It reads the data at each of the register addresses and outputs those

values to the ALU. The register module also takes in a reg_write control signal, and the address and data for the destination register currently being written to in stage 4, which will only be used if the reg_write signal is asserted from the write back stage (Stage 4). Next, stage 3 is called the Execution Stage (EX), which is the actual multimedia ALU. In this stage, the values of the registers being operated on first each pass through a multiplexor, which determines if the register values should be coming from the register file, or from the writeback stage of the pipeline. The control signal for each mux is determined by the forwarding unit, which takes in the reg_write signal and the 128 bit data value from the write back stage. During this stage, any computations necessary are performed on the register values that were read, and output is a new value to be stored in the destination register rd. The final stage (Stage 4) is the Write Back stage, (WB). This stage takes the 128 bit data to be written to rd and the 5 bit address to the specific register being written to, The WB stage also determines if the reg_write signal should be asserted or not, which in turn returns to the forwarding unit and the register file, and functions as these components were described earlier.

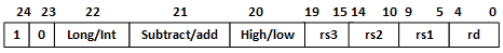
Assembler:

The format of the 25-bit machine code instructions is as follows:

Load immediate:

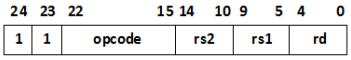


Multiply-add and multiply-subtract r4-instruction format:



- Includes simals, simahs, simsls, simshs, slmals, slmahs, slmsls, slmshs

R3-instruction format:



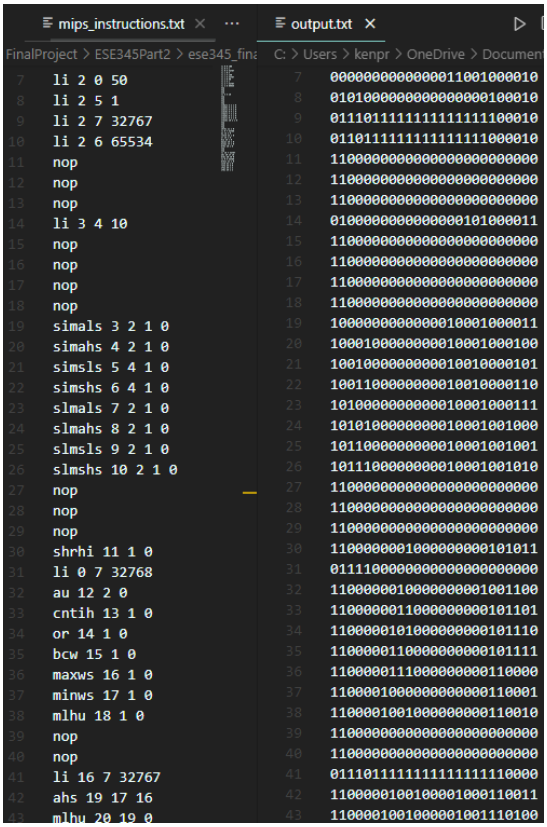
- Includes nop, shrhi, au, cntih, ahs, or, bcw, maxws, minws, mlhu, mlhss, and, invb, rotw, sfwu, sfhs

```
if instruction.startswith("li"):
  instr_list = instruction.rsplit(" ")
  mach_code = "0" + str(bin2(int(instr_list[2]))).zfill(3) + str(bin2(int(instr_list[3]))).zfill(16) + str
elif instruction.startswith("simals"):
  instr_list = instruction.rsplit(" ")
  mach_code = "10000" + str(bin2(int(instr_list[4]))).zfill(5) + str(bin2(int(instr_list[3]))).zfill(5) +
elif instruction.startswith("simahs"):
  instr_list = instruction.rsplit(" ")
  mach_code = "10001" + str(bin2(int(instr_list[4]))).zfill(5) + str(bin2(int(instr_list[3]))).zfill(5) +
elif instruction.startswith("simsls"):
  instr_list = instruction.rsplit(" ")
  mach_code = "10010" + str(bin2(int(instr_list[4]))).zfill(5) + str(bin2(int(instr_list[3]))).zfill(5) +
elif instruction.startswith("simshs"):
  instr_list = instruction.rsplit(" ")
  mach_code = "10011" + str(bin2(int(instr_list[4]))).zfill(5) + str(bin2(int(instr_list[3]))).zfill(5) +
elif instruction.startswith("slmals"):
  instr_list = instruction.rsplit(" ")
  mach_code = "10100" + str(bin2(int(instr_list[4]))).zfill(5) + str(bin2(int(instr_list[3]))).zfill(5) +
elif instruction.startswith("slmahs"):
  instr_list = instruction.rsplit(" ")
  mach_code = "10101" + str(bin2(int(instr_list[4]))).zfill(5) + str(bin2(int(instr_list[3]))).zfill(5) +
```

The assembler we wrote to get the machine code inputs for the instruction buffer was completed in python. The snippet provided shows the format of the assembler; it is simply a premade if-else list that checks for all possible instructions that could be fed into the pipeline. It converts the instructions from a pseudo MIPS code to machine code and inputs that into the buffer.

The assembly instructions can be entered into the input file on separate lines for the assembler as follows:

- Load immediate: li rd index imm
- R4-instruction: instr rd rs1 rs2 rs3
- R3-instruction: instr rd rs1 rs2



Design Procedure and Testbenches:

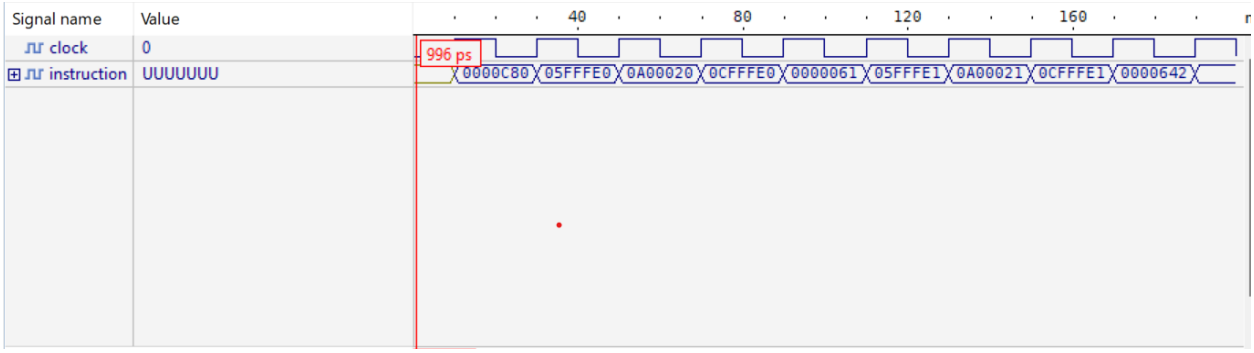
Instruction Buffer:

```
begin
    process (clock)
        file file_pointer : text;
        variable line_content : std_logic_vector (24 downto 0);
        variable line_num : line;
        variable j : integer := 0;
    begin
        file_open (file_pointer, "output.txt", READ_MODE);
        while ((not endfile(file_pointer)) and j < 64) loop
            readline (file_pointer, line_num);
            READ (line_num,line_content);
            instructions(j) <= line_content;
            j := j + 1;
        end loop;
        file_close (file_pointer);

    end process;


    process(clock)
    variable PC : integer := 0;
    begin
    if rising_edge(clock) then
        if (PC < 64) then
            instruction <= instructions(PC);
        end if;
        PC := PC + 1;
    end if;
    end process;
end behavioral;
```

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4   use work.all;
5
6   entity instruction_buffer_tb is
7   end instruction_buffer_tb;
8
9   architecture tb_architecture of instruction_buffer_tb is
10
11  -- stimulus signals
12  signal clock : std_logic := '0';
13  signal instruction : std_logic_vector (24 downto 0);
14
15  constant period : time := 10ns;
16
17  begin
18
19
20      -- Unit Under Test port map
21      UUT : entity instruction_buffer
22          port map (
23              clock => clock,
24              instruction => instruction
25          );
26
27          clk : process                -- system clock
28          begin
29              for i in 0 to 1032 loop
30                  wait for period;
31                  clock <= not clock;
32              end loop;
33              wait;
34          end process;
35
36      end tb_architecture;
```

The instruction buffer is stage 1 of the pipeline, or the instruction fetch stage. It simply reads the text file of the list of 25-bit machine code instructions and store the instructions into an array. The PC counter acts as the index of the array and increments on every clock cycle. The buffer then outputs the instruction at that index. The instruction buffer can hold up to 64 instructions. This is a very simple testbench for the instruction buffer, and the waveform shows a new instruction being passed through on every clock cycle.

Waveform:



Register Module:

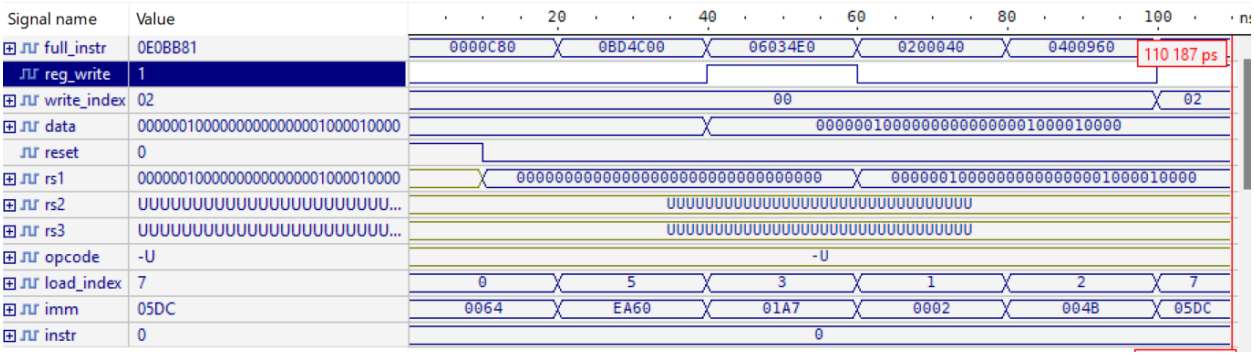
```
entity register_module is
    port (
        -- 25-bit instruction passed through the instruction buffer
        full_instr : in std_logic_vector (24 downto 0);
        -- Data to update the register file from the write back stage
        reg_write : in std_logic;
        write_index : in std_logic_vector (4 downto 0);
        data : in std_logic_vector (127 downto 0);
        reset : in std_logic;
        -- Output signals to be fed to the ALU
        rs1 : out std_logic_vector (127 downto 0);
        rs2 : out std_logic_vector (127 downto 0);
        rs3 : out std_logic_vector (127 downto 0);
        opcode : out std_logic_vector (7 downto 0);
        load_index : out std_logic_vector (2 downto 0);
        imm : out std_logic_vector (15 downto 0);
        instr : out std_logic_vector (1 downto 0);
        rs1_addr : out std_logic_vector (4 downto 0);
        rs2_addr : out std_logic_vector (4 downto 0);
        rs3_addr : out std_logic_vector (4 downto 0);
        rd_addr : out std_logic_vector (4 downto 0)
    );
end register_module;


architecture behavioral of register_module is

-- Stores the data of each of the 32 registers
type registerArray is array (0 to 31) of std_logic_vector(127 downto 0);
signal registers : registerArray := (others => (others => '0'));

begin

    process (full_instr, reset, registers, reg_write, data, write_index)
    file file_pointer : text;
    variable line_content : std_logic_vector (127 downto 0);
    variable line_num : line;

    begin
        --report "Write index : " & to_string(write_index);
        --report "Write value : " & to_string(data);
        --report "reg_write : " & to_string(reg_write);

        -- update the register from the write back only if the reg_write signal is '1'
        if reg_write = '1' then
            registers(to_integer(unsigned(write_index))) <= data;
        end if;

        -- Reset all register values to 0 if reset is '1'
        if reset = '1' then
            for i in 0 to 31 loop
                registers(i) <= std_logic_vector(to_unsigned(0, 128));
            end loop;

        end if;

        -- Identify if the instruction is either load immediate, multiply-add and multiply-subtract r4-ir
        -- or r3-instruction format. Then, decode the 25-bit instruction into its necessary fields to be
        if full_instr(24) = '0' then
            -- LOAD IMMEDIATE --
            instr <= "00";
            load_index <= full_instr(23 downto 21);
            imm <= full_instr(20 downto 5);
            rs1_addr <= full_instr(4 downto 0);
            rs1 <= registers(to_integer(unsigned(full_instr(4 downto 0))));
        else
            instr <= full_instr(24 downto 23);
            if full_instr(23) = '0' then
                -- R4 INSTRUCTIONS
                opcode <= "00000" & full_instr(22 downto 20);
                rs3_addr <= full_instr(19 downto 15);
                rs3 <= registers(to_integer(unsigned(full_instr(19 downto 15))));
                rs2_addr <= full_instr(14 downto 10);
                rs2 <= registers(to_integer(unsigned(full_instr(14 downto 10))));
                rs1_addr <= full_instr(9 downto 5);
                rs1 <= registers(to_integer(unsigned(full_instr(9 downto 5))));
            else
                -- R3 INSTRUCTIONS
                opcode <= "0000" & full_instr(18 downto 15);
                rs2_addr <= full_instr(14 downto 10);
                rs2 <= registers(to_integer(unsigned((full_instr(14 downto 10)))));
                rs1_addr <= full_instr(9 downto 5);
                rs1 <= registers(to_integer(unsigned((full_instr(9 downto 5)))));
            end if;
        end if;

        -- The address for writing the result is the same part of every 25-bit instruction format
        rd_addr <= full_instr(4 downto 0);

        -- Update the registers text file with the current values of the registers
        file_open (file_pointer, "registers.txt", WRITE_MODE);
        for i in 0 to 31 loop
            write (line_num, to_string(i) & ": " & to_string(registers(i)(127 downto 112)) & " "
            & to_string(registers(i)(111 downto 96)) & " " & to_string(registers(i)(95 downto 80)) & " "
            & to_string(registers(i)(79 downto 64)) & " " & to_string(registers(i)(63 downto 48)) & " "
            & to_string(registers(i)(47 downto 32)) & " " & to_string(registers(i)(31 downto 16)) & " "
            & to_string(registers(i)(15 downto 0)));
            writeline (file_pointer, line_num);
        end loop;
        file_close (file_pointer);

    end process;



end behavioral;
```

The register module is stage 2 of the pipeline, or the instruction decode stage, and its function can be thought of as twofold. First, it receives three inputs from stage 4 of the pipeline (writeback), including a register address, data to be written to that address, and a register write signal that must be asserted for writing to occur. The register module stores an array of 32 128-bit registers that is both read from, and written to when the reg_write signal is asserted. The data of this array is also written to a registers.txt file that can be viewed during/after simulation. The other function of the register module is reading in the 25-bit instruction and breaking it into components needed for the ALU, such as the addresses of each of the registers and reading their values, the opcode, the 2-bit field to identify the instruction format, and the immediate value and the load index for the load immediate instruction, which are all then output to stage 3.

In this testbench, we instantiate several examples of inputs including some instructions with an asserted reg_write signal and some without.

```
full_instr <= "0101111010100110000000000";
reg_write <= '0';
write_index <= "00000";
data <= "00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000";
wait for 20ns;


full_instr <= "0011000000011010011100000";
reg_write <= '1';
write_index <= "00000";
data <= "00000000000000000000000000001000000000000000000000000000000000000000000000000000000000000000000000000000000000000000";
wait for 20 ns;
```

Our waveform:



The register module also writes the register values into a text file of which a snippet is shown:

```
0: 0000000000000000 0000000000010000 0000000000000000 0000000000000000 0000000000000000 0000000000010000 0000000000000001 0000000000000000
1: 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
2: 0000000000000000 0000000000010000 0000000000000000 0000000000000000 0000000000000000 0000000000010000 0000000000000001 0000000000000000
3: 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
```

ALU:

Sample function code snippet: SignedIntegerMultiplySubtractLowWithSaturation (simsls)

```
begin
    for i in 0 to 3 loop
        reg1 :=    signed(rs1(32*i + 31 downto 32*i));
         if (reg1(31) = '1') then
             newreg1 := '1' & reg1;
         else
             newreg1 := '0' & reg1;
         end if;

         product := signed(rs3(32*i + 15 downto 32*i)) * signed(rs2(32*i + 15 downto 32*i));
         if (product(31) = '1') then
             newproduct := '1' & product;
         else
             newproduct := '0' & product;
         end if;

        result := newreg1 - newproduct;

        if result > to_signed(int_max, 33) then
            result := to_signed(int_max, 33);
        elsif result < to_signed(int_min, 33) then
            result := to_signed(int_min, 33);
        end if;

        rd_temp(32*i + 31 downto 32*i) := std_logic_vector(result(31 downto 0));
    end loop;
    return rd_temp;
end function;
```

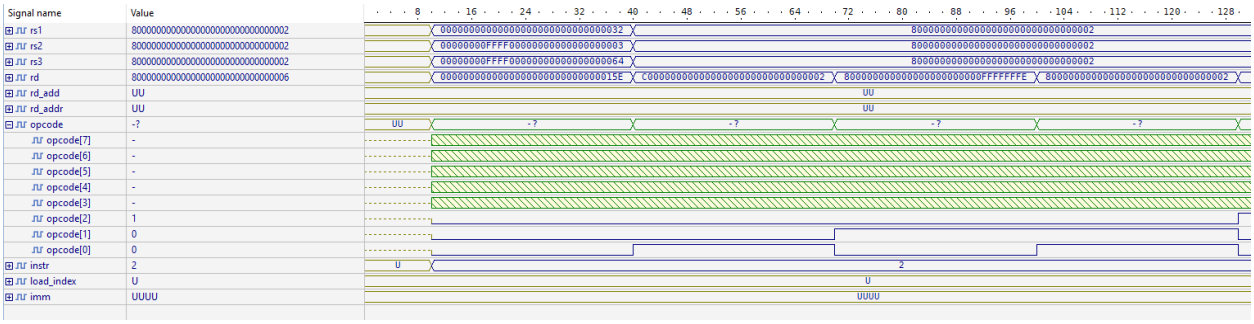
The multimedia ALU is stage 3 of the pipeline, or the execution stage, and it essentially performs any computations necessary to output the resulting register value for a given operation. The inputs are the decoded components of the instruction from stage 2, and the input register values are the values read in the register module, unless data forwarding occurs, in which case one of the values (rs1, rs2, rs3) will be replaced by the value in the writeback stage (stage 4). The above code is an example of one of the implementations, the simsls instruction.

Testbench:

In our comprehensive testbench, we set an example for every instruction of the code. We just change the instr and opcode value and keep the register values the same for simplicity’s sake.

```
wait for 10ns;

rs1 <= "10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000";
rs2 <= "10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000";
rs3 <= "10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000";
instr <= "10";
opcode <= "-----001";
wait for 20 ns;
report "rd = " & to_string(rd);
```

Waveform:



This snippet of the waveform shows the first four r4-instructions, and you can see the calculated values for rd for each instruction is correct for the given input register values.

Forwarding and Multiplexing

```
begin
    process (ctrl1, ctrl2, ctrl3, val1, val2, val3, wbval)
    begin
        if ctrl1 = '1' then
            val1_out <= wbval;
        else
            val1_out <= val1;
        end if;
        if ctrl2 = '1' then
            val2_out <= wbval;
        else
            val2_out <= val2;
        end if;
        if ctrl3 = '1' then
            val3_out <= wbval;
        else
            val3_out <= val3;
        end if;
    end process;
```

```
process(rs1_addr, rs2_addr, rs3_addr, rd_addr)
begin
if (rs1_addr = rd_addr and reg_write = '1') then
    ctrl1 <= '1';
else
    ctrl1 <= '0';
end if;

if (rs2_addr = rd_addr and reg_write = '1') then
    ctrl2 <= '1';
else
    ctrl2 <= '0';
end if;

if (rs3_addr = rd_addr and reg_write = '1') then
    ctrl3 <= '1';
else
    ctrl3 <= '0';
end if;
end process;
```

On the left is a snippet of code from our multiplexer, which checks if the values to be computed require an up-to-date value from a register that is currently being written to. If the forwarding unit on the right determines that an input register address being used in stage 3 is the same as an address currently being written to in stage 4, it will send a respective control signal to the multiplexer, which will then update the value of the necessary register(s) to achieve accurate results in computation during stage 3.

Buffers

```vhdl
-- A buffer between the instruction fetch and instruction decode stages (stages 1 & 2)
-- On each clock cycle, the instruction read from the instruction buffer is simply
-- passed through to stage 2 (the register module)

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity if_id is
    port (
        clock : in std_logic;
        instruction_in : in std_logic_vector(24 downto 0);
        instruction_out : out std_logic_vector(24 downto 0)
    );
end if_id;

architecture structural of if_id is
begin
    process(clock, instruction_in)
    begin
    if rising_edge(clock) then
        instruction_out <= instruction_in;
    end if;
    end process;
end structural;
```

```vhdl
-- A buffer between the instruction decode and execution stages (stages 2 & 3)
-- On each clock cycle, each of the output values from the decoding stage is simply
-- passed through to the ALU for execution.

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity id_ex is
    port (
        instruction : in std_logic_vector(24 downto 0);
        val1 : in std_logic_vector (127 downto 0);
        val2 : in std_logic_vector (127 downto 0);
        val3 : in std_logic_vector (127 downto 0);
        rd_add : in std_logic_vector (4 downto 0);
        opcode : in std_logic_vector (7 downto 0);
        load_index : in std_logic_vector (2 downto 0);
        imm : in std_logic_vector (15 downto 0);
        instr : in std_logic_vector (1 downto 0);
        rs1_addr, rs2_addr, rs3_addr : in std_logic_vector (4 downto 0);
        clk : in std_logic;
        rs1_add, rs2_add, rs3_add : out std_logic_vector (4 downto 0);
        instruction_out : out std_logic_vector(24 downto 0);
        opcode_out : out std_logic_vector (7 downto 0);
        load_index_out : out std_logic_vector (2 downto 0);
        imm_out : out std_logic_vector (15 downto 0);
        instr_out : out std_logic_vector (1 downto 0);
        val1_out : out std_logic_vector (127 downto 0);
        val2_out : out std_logic_vector (127 downto 0);
        val3_out : out std_logic_vector (127 downto 0);
        rd_addr : out std_logic_vector (4 downto 0)
    );
end id_ex;


architecture structural of id_ex is
begin
    process (clk, instruction, val1, val2, val3, instr, imm, load_index,
    rd_add, rs1_addr, rs2_addr, rs3_addr, opcode)
    begin
        if rising_edge(clk) then
            instruction_out <= instruction;
            val1_out <= val1;
            val2_out <= val2;
            val3_out <= val3;
            instr_out <= instr;
            imm_out <= imm;
            load_index_out <= load_index;
            rd_addr <= rd_add;
            rs1_add <= rs1_addr;
            rs2_add <= rs2_addr;
            rs3_add <= rs3_addr;
            opcode_out <= opcode;
        end if;
    end process;
```

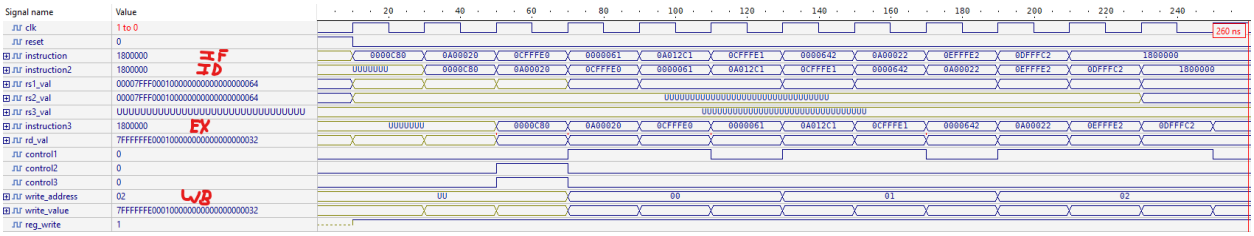
```vhdl
-- A buffer between the execution and write back stages (stages 3 & 4)
-- The resulting output register value from the ALU is passed through
-- on every clock cycle. This buffer outputs that value, along with a
-- reg_write signal and register address to write to.

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ex_wb is
    port (
        rd_add : in std_logic_vector (4 downto 0);
        instr : in std_logic_vector(24 downto 0);
        val : in std_logic_vector (127 downto 0);
        clk : in std_logic;
        val_out : out std_logic_vector (127 downto 0);
        reg_write : out std_logic;
        rd_addr : out std_logic_vector (4 downto 0)
    );
end ex_wb;


architecture structural of ex_wb is
begin
    process (clk, instr, val, rd_add)
    begin
        if rising_edge(clk) then
            --report "rd_addr : " & to_string(rd_add);
            --report "val : " & to_string(val);
            --report "instr : " & to_string(instr);
            val_out <= val;
            rd_addr <= rd_add;
            -- will only not write new value if the instruction is nop
            reg_write <= '0' when instr = "11000000000000000000000000" else '1';
        end if;
    end process;


end structural;
```

The above buffers are used to address the timing issues in having multiple stages. These buffers essentially will read in the output from a previous stage, and on each clock cycle (rising edge), they will output those values to the next stage. For example, the instruction that is output from the instruction buffer (stage 1) can only be read into stage 2 on a rising clock edge, since the IF/ID buffer lies between them, and will only send the value once per clock cycle.

Four-Stage Pipelined Multimedia ALU

```
begin

    -- input clk signal, output instruction goes to if_id
    u1 : instruction_buffer port map (clock => clk, instruction => instr_buff);

    -- input clk signal, input instruction from instruction_buffer, output instruction goes to register_module
    u2 : if_id port map (clock => clk, instruction_in => instr_buff, instruction_out => instr_reg);

    -- input instruction from if_id, input data for writing comes from ex_wb, outputs decoded instruction
    u3 : register_module port map (full_instr => instr_reg, reg_write => rw, write_index => write_addr,
        data => write_val, reset => rst, rs1 => r1, rs2 => r2, rs3 => r3, opcode => opc,
        imm => imm, load_index => load_in, instr => instr_id, rs1_addr => r1_addr,
        rs2_addr => r2_addr, rs3_addr => r3_addr, rd_addr => rd_add);

    -- inputs decoded instruction from register_module, outputs decoded instruction to multimedia_alu.
    -- outputs rs1, rs2, rs3 addresses to forwarding unit and values to multiplexer
    u4 : id_ex port map (clk => clk, instruction => instr_reg, val1 => r1, val2 => r2, val3 => r3,
        rd_add => rd_add, opcode => opc, load_index => load_in, imm => imm, instr => instr_id,
        rs1_addr => r1_addr, rs2_addr => r2_addr, rs3_addr => r3_addr, instruction_out => instr_wb,
        val1_out => val1, val2_out => val2, val3_out => val3, rd_addr => rd_addr, rs1_add => r1_addr_1,
        rs2_add => r2_addr_1, rs3_add => r3_addr_1, opcode_out => opc_1, load_index_out => load_in_1,
        imm_out => imm_1, instr_out => instr_id_1);

    -- inputs ctrl signals from forwarding_unit, values 1-3 from id_ex, wb value from ex_wb
    -- outputs rs1, rs2, rs3 values to be used in multimedia_alu
    u5 : mux port map (ctrl1 => ctrl1, ctrl2 => ctrl2, ctrl3 => ctrl3, val1 => val1, val2 => val2,
    val3 => val3, wbval => write_val, val1_out => val1_1, val2_out => val2_1, val3_out => val3_1);

    -- inputs rs1, rs2, rs3 addresses from id_ex, wb address and reg_write from ex_wb
    -- outputs multiplexer control signals
    u6 : forwarding_unit port map (rs1_addr => r1_addr_1, rs2_addr => r2_addr_1, rs3_addr => r3_addr_1,
    rd_addr => write_addr, reg_write => rw, ctrl1 => ctrl1, ctrl2 => ctrl2, ctrl3 => ctrl3);

    -- inputs rs1, rs2, rs3 values from multiplexer, other decoded parts of instruction from id_ex
    -- outputs the computed register value to be used in ex_wb
    u7 : multimedia_alu port map (rs1 => val1_1, rs2 => val2_1, rs3 => val3_1, opcode => opc_1,
    load_index => load_in_1, imm => imm_1, instr => instr_id_1, rd => rd);

    -- inputs register value from multimedia_alu, the address to be written to from id_ex
    -- outputs the value to be written, the address to write to, and the reg_write signal, all
    -- to be used in the register_module
    u8 : ex_wb port map (clk => clk, val => rd, instr => instr_wb, rd_add => rd_addr, val_out => write_val,
        rd_addr => write_addr, reg_write => rw);

    -- Stage 1
    instruction <= instr_buff;

    -- Stage 2
    instruction2 <= instr_reg;
    rs1_val <= r1;
    rs2_val <= r2;
    rs3_val <= r3;

    -- Stage 3
    control1 <= ctrl1;
    control2 <= ctrl2;
    control3 <= ctrl3;
    instruction3 <= instr_wb;
    rd_val <= rd;

    -- Stage 4
    reg_write <= rw;
    write_address <= write_addr;
    write_value <= write_val;

    rst <= reset;
```

The above snippet is of our source code for the overall four-staged pipelined multimedia unit. You can see the mappings of all the ports to many different signals to connect each component of this project to one another. At the bottom, you can see the output ports of this unit which are then used to illustrate the waveform shown below.



As you can see from the image above, we have a pipelined multimedia unit with four stages. Each instruction fed into the instruction buffer at the first stage is stepped through the next three stages, one per clock cycle.

```
    -- Unit Under Test port map
    UUT : entity four_stage_pipeline
        port map (
            clk => clk,
            reset => reset,
            instruction => instruction,
            instruction2 => instruction2,
            instruction3 => instruction3,
            control1 => control1,
            control2 => control2,
            control3 => control3,
            rs1_val => rs1_val,
            rs2_val => rs2_val,
            rs3_val => rs3_val,
            rd_val => rd_val,
            write_address => write_address,
            write_value => write_value,
            reg_write => reg_write
        );

        clock : process                 -- system clock
        file file_pointer : text;
        variable file_is_open : boolean := false;
        variable line_content : std_logic_vector (127 downto 0);
        variable line_num : line;
        variable j : integer := 1;
        begin
            reset <= '1', '0' after 10ns;
            for i in 0 to 1032 loop
                wait for period;
                clk <= not clk;
                if clk = '1' then
                    if not file_is_open then
                        file_open (file_pointer, "results.txt", WRITE_MODE);
                        file_is_open := true;
                    end if;

                    write (line_num, "Cycle: " & to_string(j) & ":");
                    writeline (file_pointer, line_num);

                    write (line_num, "    Stage 1:  instr = " & to_string(instruction));
                    writeline (file_pointer, line_num);

                    write (line_num, "    Stage 2:  instr = " & to_string(instruction2));
                    writeline (file_pointer, line_num);
                    write (line_num, "              rs1 = " & to_string(rs1_val));
                    writeline (file_pointer, line_num);
                    write (line_num, "              rs2 = " & to_string(rs2_val));
                    writeline (file_pointer, line_num);
                    write (line_num, "              rs3 = " & to_string(rs3_val));
                    writeline (file_pointer, line_num);

                    write (line_num, "    Stage 3:  instr = " & to_string(instruction3));
                    writeline (file_pointer, line_num);
                    write (line_num, "              rd = " & to_string(rd_val));
                    writeline (file_pointer, line_num);
                    if (control1 = '1' or control2 = '1' or control3 = '1') then
                        write (line_num, "              Data forwarding occurs");
                        writeline (file_pointer, line_num);
                    end if;

                    write (line_num, "    Stage 4:  reg_write = " & to_string(reg_write));
                    writeline (file_pointer, line_num);
                    write (line_num, "              val = " & to_string(write_value));
                    writeline (file_pointer, line_num);
                    write (line_num, "              addr = " & to_string(write_address));
                    writeline (file_pointer, line_num);

                    write (line_num, " ");
                    writeline (file_pointer, line_num);

                    j := j + 1;
                end if;
            end loop;
            wait;
```

Above is the testbench written to verify the completed four stage pipelined multimedia ALU. It simply generates a clock waveform to be used and writes relevant values to a results.txt file, which can be seen below.

```
Cycle: 4:
    Stage 1:  instr = 00000000000000000001100001
    Stage 2:  instr = 01100111111111111100000
              rs1 = 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001100100
              rs2 = UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
              rs3 = UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
    Stage 3:  instr = 01010000000000000100000
              rd = 00000000000000000000000000000000000000000000000000000000000000010000000000000000000000000000000000000000000000000001100100
             Data forwarding occurs
    Stage 4:  reg_write = 1
              val = 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001100100
              addr = 00000

Cycle: 5:
    Stage 1:  instr = 01010000000100101100001
    Stage 2:  instr = 00000000000000000001100001
              rs1 = 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
              rs2 = UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
              rs3 = UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
    Stage 3:  instr = 01100111111111111100000
              rd = 00000000000000000111111111111110000000000000100000000000000000000000000000000000000000000000000000000000001100100
             Data forwarding occurs
    Stage 4:  reg_write = 1
              val = 00000000000000000000000000000000000000000000100000000000000000000000000000000000000000000000000000000000001100100
              addr = 00000

Cycle: 6:
    Stage 1:  instr = 01100111111111111100001
    Stage 2:  instr = 01010000000100101100001
              rs1 = 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
              rs2 = UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
              rs3 = UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
    Stage 3:  instr = 00000000000000000001100001
              rd = 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011
    Stage 4:  reg_write = 1
              val = 00000000000000000111111111111110000000000000100000000000000000000000000000000000000000000000000000000000001100100
              addr = 00000

Cycle: 7:
    Stage 1:  instr = 00000000000011001000010
    Stage 2:  instr = 01100111111111111100001
              rs1 = 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011
              rs2 = UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
              rs3 = UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
    Stage 3:  instr = 01010000000100101100001
              rd = 00000000000000000000000000000000000000001001011000000000000000000000000000000000000000000000000000000000000000000011
             Data forwarding occurs
    Stage 4:  reg_write = 1
              val = 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011
              addr = 00001
```

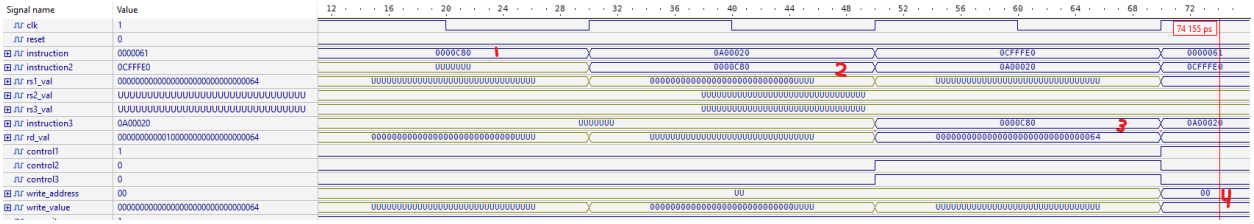
Above is the results file produced by the testbench for the multimedia unit, which shows the status of each individual stage at every clock cycle. The first stage shows the 25-bit input instruction into the IF/ID buffer. The second stage shows the instruction at this point, along with all the input register values. The third stage shows the instruction at this point, along with the resulting register value after performing some ALU operation. The fourth stage shows the write enable signal, the value to be written to the register file, and the address of the register.

```
0: 10000000000000 0111111111111111 10000000000000 00000000000000 00000000000000 00000000000000 00000000000000 00000000001100100
1: 00000000000000 0111111111111111 00000000010010110 00000000000000 00000000000000 00000000000000 00000000000000 00000000000000011
2: 01111111111111 1111111111111110 00000000000000001 00000000000000 00000000000000 00000000000000 00000000000000 00000000000110010
3: 01111111111111 1111111111111111 00000000000000001 00000000000000 00000000000000 00000000000000 00000000000000 00000000101011110
4: 01111111111111 1111111111111110 00000000000000001 00000000010010110 00000000000000 00000000000000 00000000000000 00000000000110010
5: 01000000000000 1111111111111101 00000000000000001 00000000010010110 00000000000000 00000000000000 11111111111111 11111111100000110
6: 01111111111111 1111111111111110 00000000000000001 00000000000000 00000000000000 00000000000000 00000000000000 00000000000110010
7: 01111111111111 1111111111111111 11111111111111111 11111111111111 00000000000000 00000000000000 00000000000000 00000000101011110
8: 01111111111111 1111111111111110 01000000000000 00000000000001 00000000000000 00000000000000 00000000000000 00000000000110010
9: 01111111111111 1111111101101000 00000000000000001 00000000000000 11111111111111 11111111111111 11111111111111 11111111100000110
10: 01111111111111 1111111111111101 11000000000000001 11111111111111 00000000000000 00000000000000 00000000000000 00000000000110010
11: 00000000000000 0000011111111111 00000000000001001 00000000000000 00000000000000 00000000000000 00000000000000 00000000000000000
12: 00000000000000 0111111111111101 00000000000000010 00000000000000 00000000000000 00000000000000 00000000000000 00000000010010110
13: 00000000000000 0000000000001111 00000000000000100 00000000000000 00000000000000 00000000000000 00000000000000 00000000000000010
14: 10000000000000 0111111111111111 00000000010010111 00000000000000 00000000000000 00000000000000 00000000000000 00000000001100111
15: 00000000000000 0000000000000011 00000000000000000 00000000000011 00000000000000 00000000000011 00000000000000 00000000000000011
16: 01111111111111 0111111111111111 00000000010010110 00000000000000 00000000000000 00000000000000 00000000000000 00000000001100100
17: 10000000000000 0111111111111111 00000000000000001 00000000000000 00000000000000 00000000000000 00000000000000 00000000000000011
18: 00111111111111 0000000000000001 00000000000000000 00000000000000 00000000000000 00000000000000 00000000000000 00000000100101100
19: 11111111111111 0111111111111111 00000000010010111 00000000000000 00000000000000 00000000000000 00000000000000 00000000001100111
20: 00111111111111 0000000000000001 00000000000000000 00000000000000 00000000000000 00000000000000 00000000000000 00010100000111100
21: 00000000000000 0111111111111111 11111111101101010 00000000000000 00000000000000 00000000000000 00000000000000 00000000000000011
22: 11111111111111 1000000000000000 11111111101101001 11111111111111 11111111111111 11111111111111 11111111111111 11111111111100
23: 11111110111111 1111111111111111 00000000000000001 00000000000000 00000000000000 00000000000000 01111000000000 00000000000000101
24: 01111111111111 0111111111111111 11111111101101011 00000000000000 00000000000000 00000000000000 00000000000000 00000000000101111
25: 01111111111111 1000000000000000 11111111101101011 00000000000000 00000000000000 00000000000000 00000000000000 00000000000101111
26: 00000000000000 0000000000000000 00000000000000000 00000000000000 00000000000000 00000000000000 00000000000000 00000000000000000
27: 00000000000000 0000000000000000 00000000000000000 00000000000000 00000000000000 00000000000000 00000000000000 00000000000000000
28: 00000000000000 0000000000000000 00000000000000000 00000000000000 00000000000000 00000000000000 00000000000000 00000000000000000
29: 00000000000000 0000000000000000 00000000000000000 00000000000000 00000000000000 00000000000000 00000000000000 00000000000000000
30: 00000000000000 0000000000000000 00000000000000000 00000000000000 00000000000000 00000000000000 00000000000000 00000000000000000
31: 00000000000000 0000000000000000 00000000000000000 00000000000000 00000000000000 00000000000000 00000000000000 00000000000000000
```

Above is the register file produced to show the state of each of the 32 registers at the end of all operations. This file can be compared with an expected results file to verify that everything is working as intended.

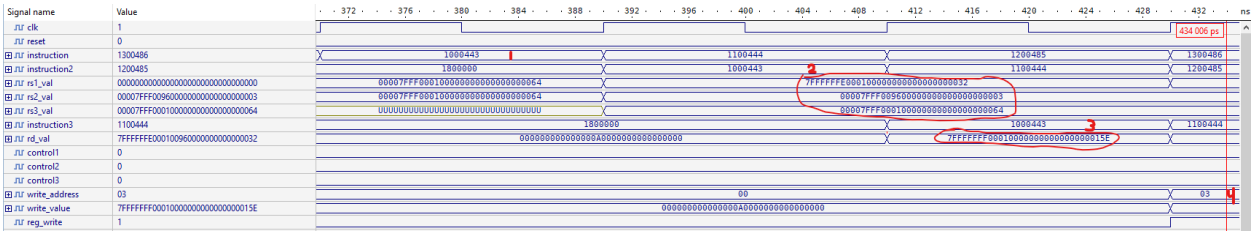
Instruction execution:

Load Immediate: li 0 0 100



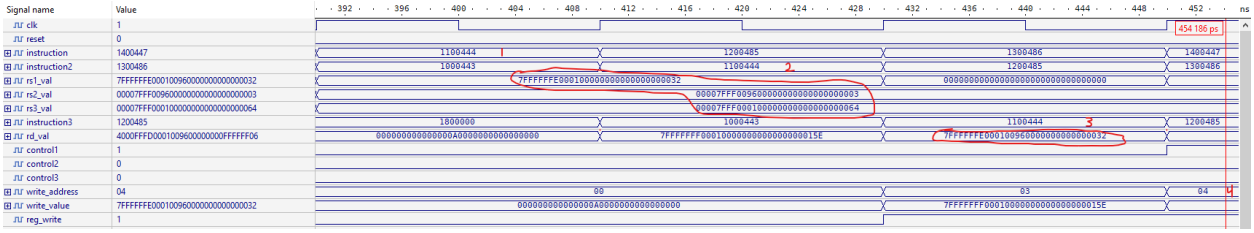
As you can see, the first load immediate instruction loads the value 100 (0x64) into halfword at index 0 of register 0. The instruction moves through the 4 stages, computing the correct register value in stage 3 and writing the value to the correct register.

Signed Integer Multiply-Add Low with Saturation: simals 3 2 1 0



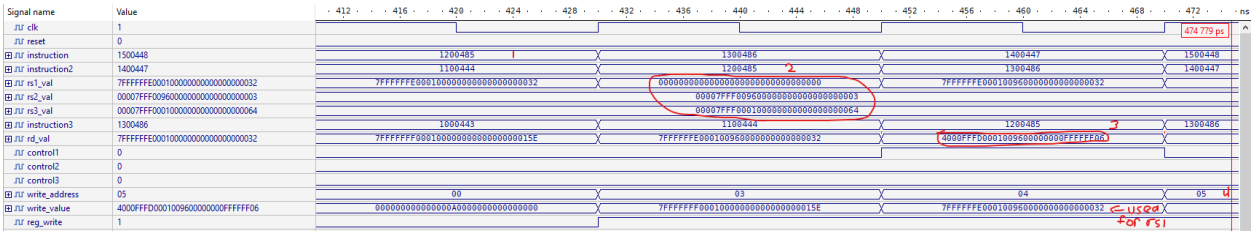
This operation should read the values from registers 0, 1, and 2, multiply the low 16 bits of every word field in registers 0 and 1, add the resulting word to the respective word fields of register 2, and store the result in register 3. For example, using the least significant word, 100*3+50 = 350, which is 0x15E.

Signed Integer Multiply-Add High with Saturation: simahs 4 2 1 0



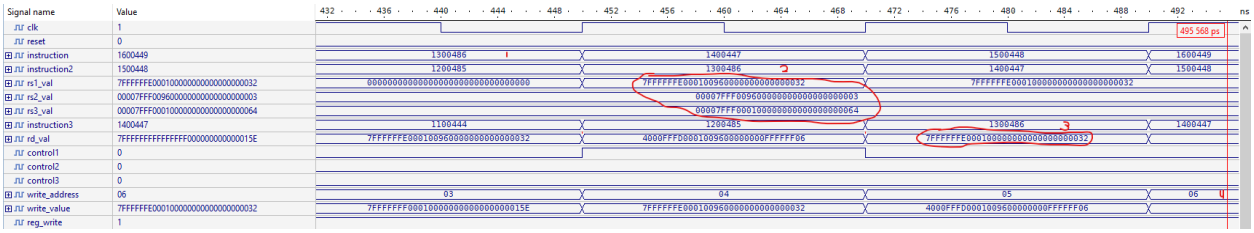
Using the same input registers and still looking at the least significant word, the high 16 bits in registers 0 and 1 are both 0, and the value of the word in register 2 is 50, so 0*0+50 = 50 (0x32).

Signed Integer Multiply-Subtract Low with Saturation: simsls 5 4 1 0



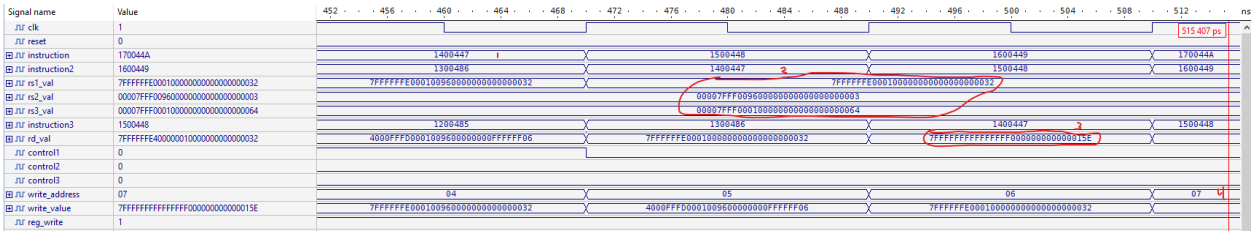
Using the input registers rs1=4, rs2=1, rs3=0, data forwarding occurs since register 4 is currently being written to from the previous instruction. While looking at the least significant word, the low 16 bits of rs2 is 3, rs3 is 100, and rs1 is 50. The result is which is 50-100*3 = -250 (0xFFFFFF06).

Signed Integer Multiply-Subtract High with Saturation: simshs 6 4 1 0



Using the same input registers as the previous instruction, and still looking at the least significant word for each register, the high 16 bits or registers 0 and 1 are 0, so the result is 50-0*0 = 50 (0x32).

Signed Long Integer Multiply-Add Low with Saturation: slmals 7 2 1 0

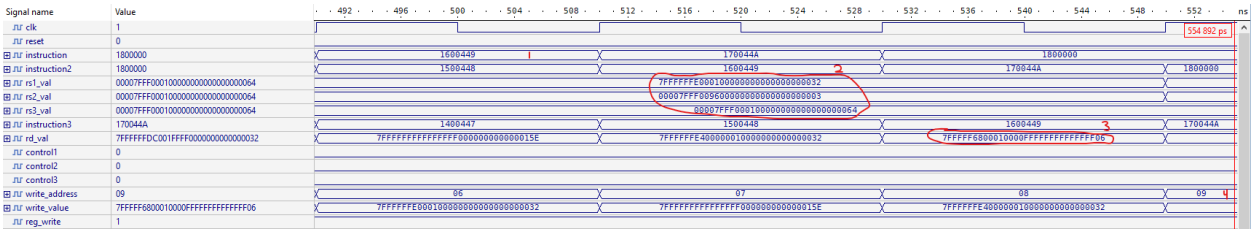


Using these input registers and looking at the most significant 64-bit fields, the result of $(9*16^5+6*16^4)*(1*16^4)+(0x7FFFFFFE00010000)$ is greater than the maximum positive integer value of a 64-bit signed integer, so it is saturated to 0x7FFFFFFFFFFFFFFF.

Signed Long Integer Multiply-Add High with Saturation: slmahs 8 2 1 0

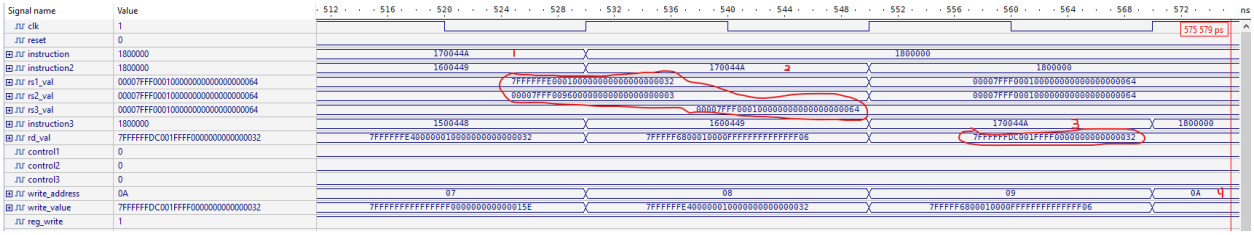
Same inputs, looking at the low 64-bit fields, since the high 32 bits in registers 0 and 1 are 0, the result is 50+0*0 = 50 (0x32).

Signed Long Integer Multiply-Subtract Low with Saturation: slmsls 9 2 1 0



Looking at the low 64-bit fields again, the low 32 bits in registers 0 and 1 are 100 and 3, respectively, and the 64-bit field in register 2 is 50. The result is 50-100*3 = -250 (0xFFFFFFFFFFFFFF06).

Signed Long Integer Multiply-Subtract High with Saturation: slmshs 10 2 1 0



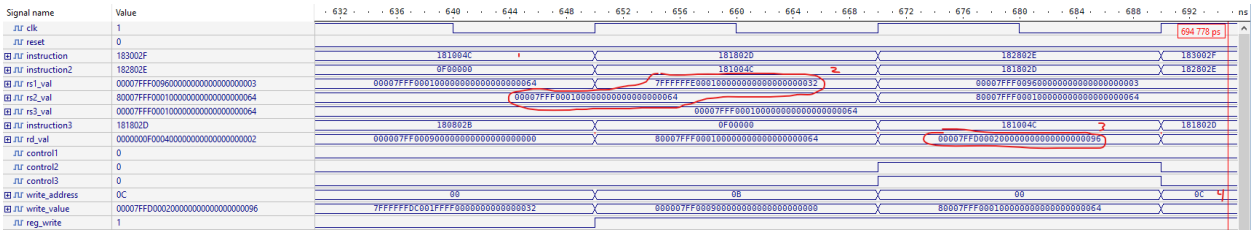
Looking at the high 64-bit fields this time, the high 32 bits in both registers 0 and 1 are 32767,

and the 64-bit field in register 2 is 0x7FFFFFFE00010000, and the result of (0x7FFFFFFE00010000)-32767*32767 = 0x7FFFFFFDC001FFFF.

Shift Right Halfword Immediate: shrhi 11 1 0

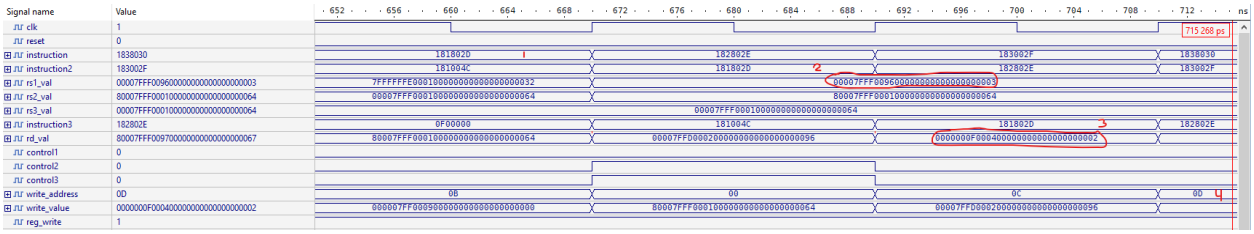
The least significant 4 bits of register 0 are 0b0100 which is 4. Each halfword in register 1 should be shifted to the right by 4 bits. For example, the least significant halfword is 0x0003, so after shifting is 0x0000. Looking at the second most significant halfword, 0x7FFF, is shifted to become 0x07FF.

Add Word Unsigned: au 12 2 0



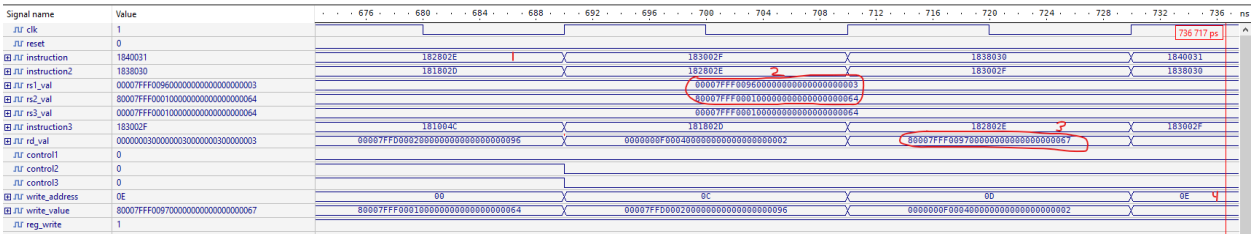
This is a standard addition of 32-bit fields. Looking at the least significant word, the sum of 50 (0x32) and 100 (0x64) is 150 (0x96).

Count 1's in halfword: cntih 13 1 0



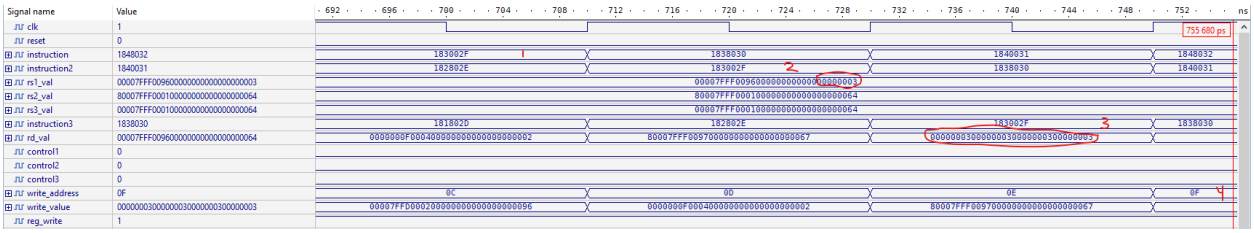
Only register 1 is used for this function as rs1, and the number of '1' bits are counted for each halfword, and the count is stored in the corresponding word in register 13. For instance, the least significant halfword is 0x0003 or 0b0000000000000011 which contains two '1' bits, so the value 2 (0x0002) is stored in the resulting register.

Bitwise or: or 14 1 0



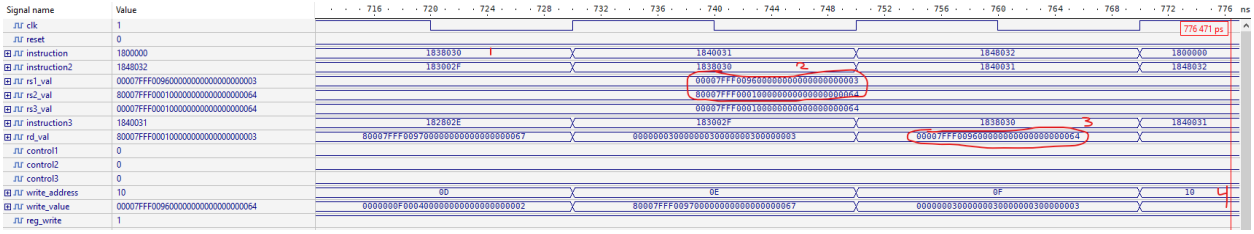
This is a standard bitwise or function, where all of the bits in register 1 is ORed with register 0, and you can see the resulting register value is correct.

Broadcast word: bcw 15 1 0



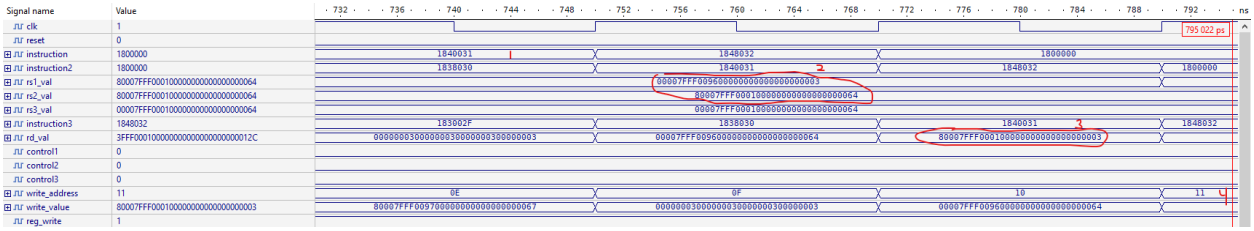
The least significant word of register 1 (0x00000003) is broadcasted to all 4 words in register 15.

Max signed word: maxws 16 1 0



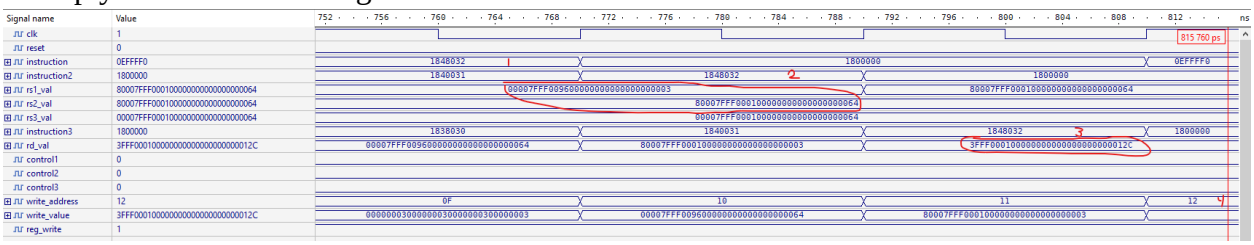
For each of the 4 words in registers 1 and 0, the maximum value is stored in the corresponding word in register 16. For example, the least significant word in register 1 holds 3 and register 0 holds 100, so the value stored in register 16 is 100.

Min signed word: minws 17 1 0



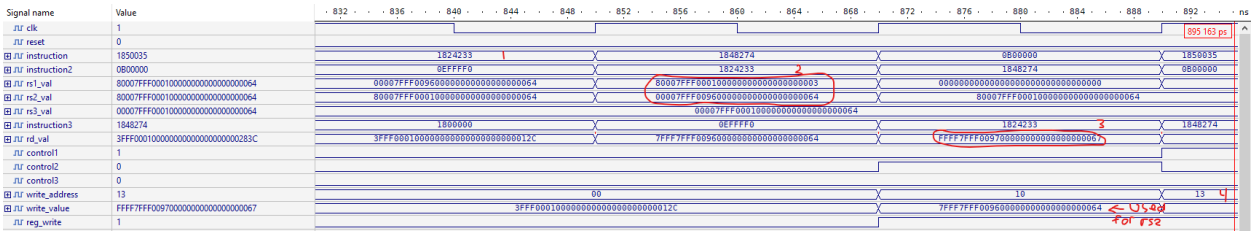
Similar to maxws, it takes the minimum value for each word between the two registers, and in the case of the least significant word again, 3 is the minimum of 3 and 100 so register 16 stores the value 3.

Multiply low halfword unsigned: mlhu 18 1 0



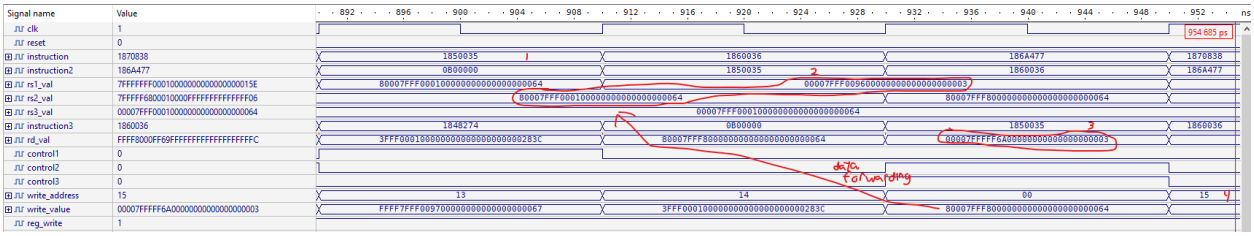
The low 16 bits of each word in registers 0 and 1 are multiplied and stored in the corresponding word in register 18. For example, the least significant word, registers 0 and 1 store values 100 and 3 respectively, so the product is 300 (0x12C).

Add halfword saturated: ahs 19 17 16



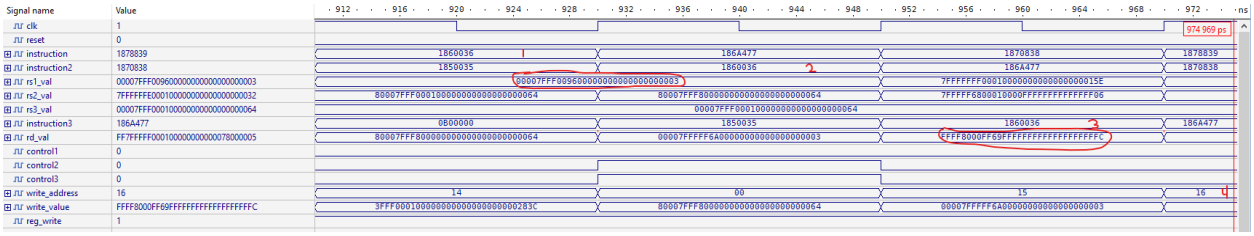
Since there was a load immediate instruction for register 16 immediately before this, data forwarding occurs for rs2. The halfwords of registers 16 and 17 are summed and stored in register 19. The least significant halfwords store values 100 and 3, so the sum is 103 (0x67). In the case of the second most significant halfwords, the sum of 32767 + 32767 exceeds the maximum value for a 16-bit signed value, so it is saturated to 0x7FFF.

Multiply by sign saturated: mlhss 21 1 0



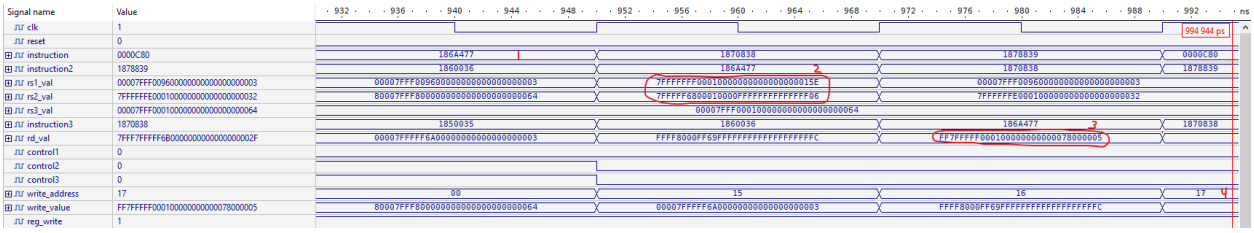
The value of each halfword in register 1 is multiplied by the sign of the corresponding halfword in register 0. For the second most significant halfword, 0x7FFF is multiplied by a positive sign, so the result is 0x7FFF. For the third most significant halfword, 150 (0x0096) is multiplied by a negative sign and becomes -150 (0xFF6A).

Invert bits: invb 22 1 0



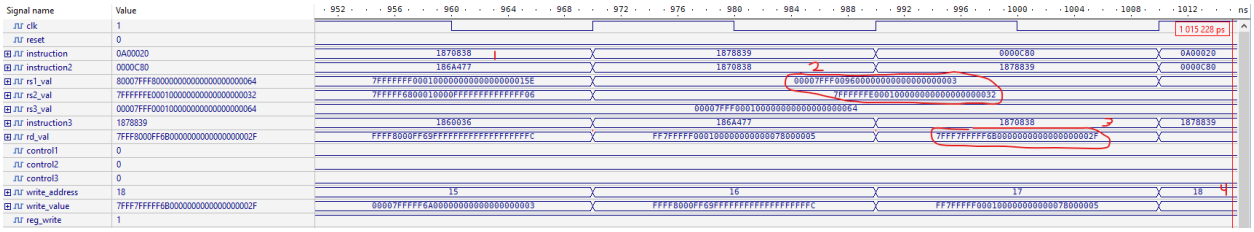
This simply flips all of the bits in register 1 and stores the result in register 22.

Rotate bits in word: rotw 23 3 9



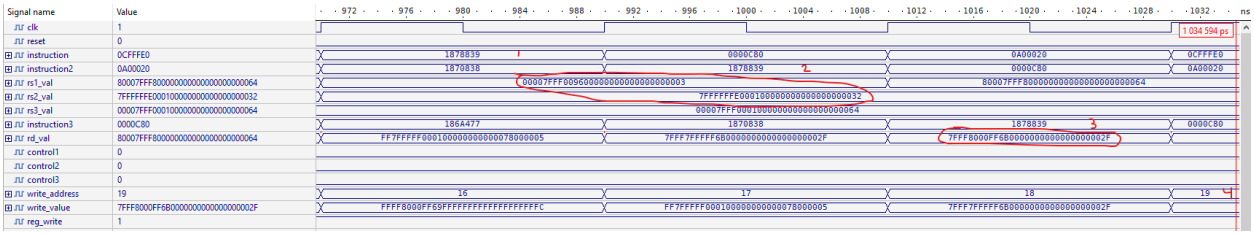
This takes the least significant 5 bits of every word in register 9 and rotates the bits in the corresponding word in register 3 by that amount to the right and stores the result in 23. The least significant word in register 3 is 0x0000015E or 0b00000000000000000000000101011110. The least significant 5 bits in the corresponding word in 9 is 00110 or 6. Rotating this by 6 bits, you get 0b01111000000000000000000000000101 or 0x78000005.

Subtract from word unsigned: sfwu 24 1 2



This subtracts the contents of each word in register 2 by the contents in the corresponding word in register 1, and stores the result in the corresponding word in register 24. Looking at the least significant word, register 2 holds 50 (0x32) and register 1 holds 3 (0x03), so the result is 47 (0x2F).

Subtract from halfword saturated: sfhs 25 1 2



This subtracts the signed halfwords of register 2 by register 1 and stores the result in the corresponding halfword in register 25. You can see saturation occurs in the second most significant halfword, where -2 is subtracted by 32767, which is less than the minimum value of a 16-bit signed integer, so the result is 0x8000. Looking at the least significant halfword, 50 (0x32) subtracted by 3 (0x03) is 47 (0x2F).

Conclusion:

We were able to verify that all of our individual components of the four stage pipeline function properly and do so for a wide variety of inputs including edge cases as well. When we put each of these components together for the pipeline, we initially found that some errors started to appear. We recognized that calling nop instructions immediately after some other instruction affected that other instruction from being performed. The example of this that we noticed was when we called nop after a li instruction, that the li instruction never wrote back to the register. The problem seemed to have been a timing issue in the register module, as adding in a couple of other relevant signals to the sensitivity list of the process fixed this issue. It also seemed as though the order of our instructions also comes into play as if the same few li instructions are reordered, the resulting registers in the register file will have different values, which did not make sense because the logic is the same. Another issue we ran into was due to attempting to use output signals as an input after assigning a value to that output in the register module. Signals in VHDL, however, are not necessarily able to be treated like variables in many other programming languages. The final issue that needed to be addressed to see the entire design work as intended was using the std_match function in the ALU when checking the opcode input. Initially, we were just using the '=' operator to compare and opcode to a vector which included "don't care" values. This did not work, as the '=' operator expects exact input to match, including "don't cares" in the specified locations, whereas, std_match will allow any value in the input vector to match with a "don't care." Finally, we were able to see the entire pipelined processor perform operations as intended, with all timing on operations working perfectly, including data forwarding for registers being modified and then read, in back-to-back instructions.