ESE 345 Final Project Report

Alan George (114484206) and Matthew Leo (114238245)

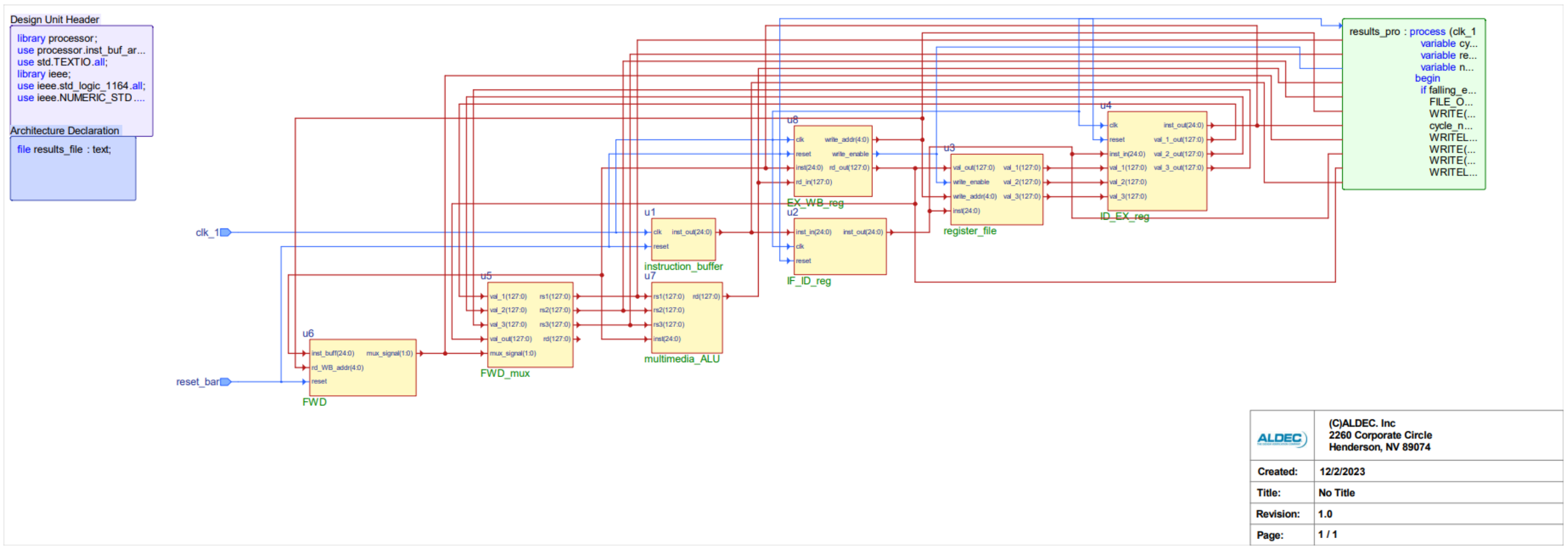
Table of Contents:

Goals

The goal of this project was to create a 4-stage pipelined multimedia unit that included Instruction Fetch, Instruction Decode, Execution, and Write Back stages. The system needs to be capable of data forwarding to prevent data hazards and limit the need for pipeline stalls. The Register File in the Instruction Decode stage needs to be capable of storing 32 128-bit registers that can be written to and stored. The Instruction Buffer is expected to store up to 64 25-bit instructions that will be inputted into the pipeline each cycle. The multimedia ALU is able to process standard instructions such as AND and OR with multiple registers, along with also computing SIMD instructions that are similar to those found in the Sony Cell SPU and Intel SSE architectures.

Each component in the processor should also be written as a behavioral model in VHDL along with one structurally-written VHDL file that connects all of the components together with relevant signals. Our project also needs to take in a text file that has the binary machine code for all 64 instructions, and place them into the Instruction Buffer before reading in any instructions. We also need to produce a “Results File” that represents the status of the pipeline at each stage, by printing the instruction being processed in each stage of the pipeline, any relevant input operands, outputs, and control signals from each stage. Finally, our system should be efficient and able to execute each instruction with the minimal amount of cycles required.

Multimedia Unit Block Diagram



Design Unit Header

```
library processor;
use processor.inst_buf_ar...
use std.TEXTIO.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.NUMERIC_STD....
```

Architecture Declaration

```
file results_file : text;
```

```
results_pro : process (clk_1
        variable cy...
        variable re...
        variable n...
      begin
        if falling_e...
          FILE_O...
          WRITE(...
          cycle_n...
          WRITEL...
          WRITE(...
          WRITE(...
          WRITEL...
```

ALDEC	(C)ALDEC. Inc 2260 Corporate Circle Henderson, NV 89074
Created:	12/2/2023
Title:	No Title
Revision:	1.0
Page:	1 / 1

Design Procedure

Register File:

After the first part of the project, we decided to start with the register file. The register file was stated to be capable of holding 32 128-bit registers so we designed an array to hold all 32 registers.

```
architecture behavioral of register_file is

    type reg_file is array (0 to 31) of std_logic_vector(127 downto 0);
    signal sig_reg_file : reg_file := (others => (others => '0'));

begin
```

These registers would be accessed using the register addresses held in the 25-bit instruction. To differentiate between the R3, R4, and Load immediate, we made sure to use the codes given at the beginning of the instruction this way we could avoid mixing up immediates, opcodes, and rs3.

```
if inst(24) = '0' then                                                  -- li instruction
    val_1 <= sig_reg_file(to_integer(unsigned(inst(4 downto 0))));           -- rd
                                                               -- *** val_1 => rs1, val_1 => rd (in port_map to ALU)
else
    -- reading from register file
    val_1 <= sig_reg_file(to_integer(unsigned(inst(9 downto 5))));      -- rs1
    val_2 <= sig_reg_file(to_integer(unsigned(inst(14 downto 10))));    -- rs2
    val_3 <= sig_reg_file(to_integer(unsigned(inst(19 downto 15))));    -- rs3

end if;

if inst(9 downto 5) = write_addr then
    val_1 <= val_out;
elsif inst(14 downto 10) = write_addr then
    val_2 <= val_out;
elsif inst(19 downto 15) = write_addr and inst(24 downto 23) = "10" then -- R4 inst only
    val_3 <= val_out;
end if;
```

Since we can have up to 3 register addresses stored in an instruction at a time, we created 3 outputs: *val_1*, *val_2*, and *val_3*. Using the addresses stored in the instruction we can access the stored data in the registers in the array and assign their 128-bit values to our outputs such as in our code shown above. (outputs shown below)

```
entity register_file is
     port(
         val_out : in STD_LOGIC_VECTOR(127 downto 0);           -- value that we are writing to a register
         write_enable : in std_logic;                           -- writing binary signal
         write_addr : in std_logic_vector(4 downto 0);          -- address of write-back register (rd)

         inst : in std_logic_vector(24 downto 0);           -- IF/ID Reg (will pass inst to ALU)

         val_1 : out STD_LOGIC_VECTOR(127 downto 0);        -- val from rs1
         val_2 : out STD_LOGIC_VECTOR(127 downto 0);        -- val from rs2
         val_3 : out STD_LOGIC_VECTOR(127 downto 0)         -- val from rs3
         );
end register_file;
```

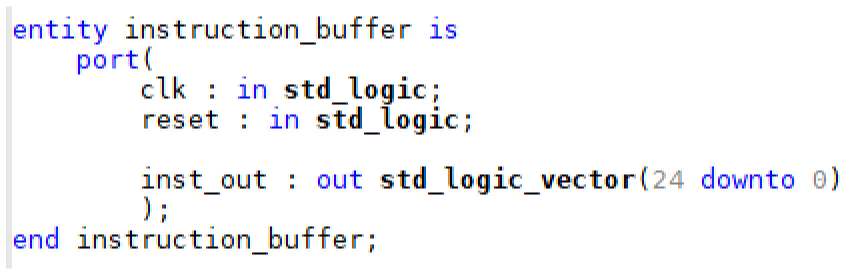
Lastly, the register file is supposed to incorporate write back features so we can adjust the values stored in the array. For that reason, we designed two more inputs: *write_enable* and *write_addr*. As the name suggests, *write_enable* signifies when the system would write back to the register file and *write_addr* stores the 5-bit address of rd in the instruction so we know where to store the updated values from the writeback stage.

Instruction Buffer:

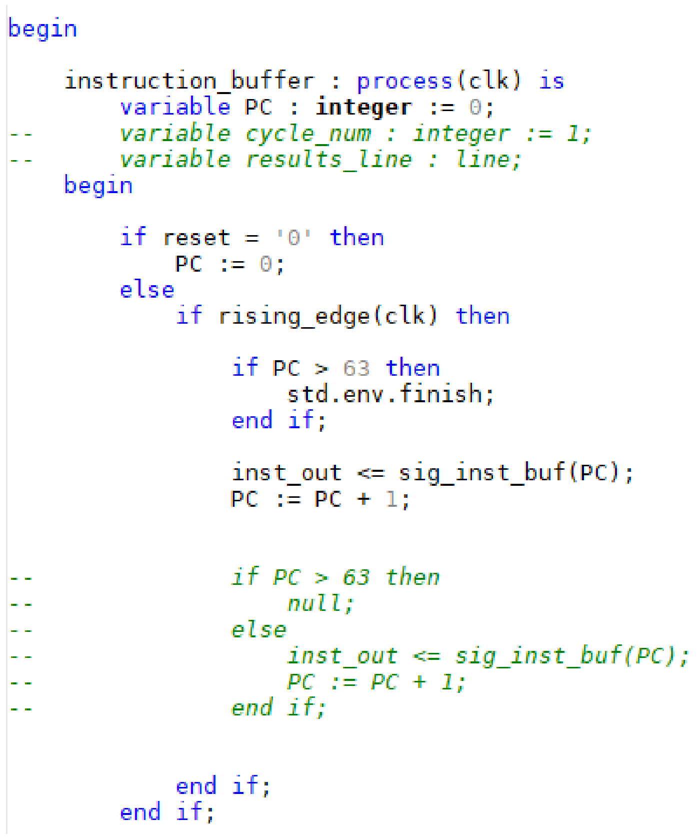
The next part we designed was the instruction buffer. The instruction buffer is loaded with 64 25-bit instructions that are given by the testbench and continuously fed into the system each clock cycle. To implement this we used a package created by the testbench to transfer instructions over to the instruction buffer. The instructions are stored in a 64 wide array each storing a 25-bit *std_logic_vector*.

```
package inst_buf_array is
    type inst_buf is array (0 to 63) of std_logic_vector(24 downto 0);      -- inst_buff array input
    shared variable sig_inst_buf : inst_buf;                        -- ALL 64 instructions in instruction buffer
```

The buffer has a *clk* input which makes sure that the instructions only get loaded each cycle.



```vhdl
entity instruction_buffer is
    port(
        clk : in std_logic;
        reset : in std_logic;

        inst_out : out std_logic_vector(24 downto 0)
        );
end instruction_buffer;
```



```vhdl
begin

    instruction_buffer : process(clk) is
        variable PC : integer := 0;
--      variable cycle_num : integer := 1;
--      variable results_line : line;
    begin

        if reset = '0' then
            PC := 0;
        else
            if rising_edge(clk) then

                if PC > 63 then
                    std.env.finish;
                end if;

                inst_out <= sig_inst_buf(PC);
                PC := PC + 1;


--              if PC > 63 then
--                  null;
--              else
--                  inst_out <= sig_inst_buf(PC);
--                  PC := PC + 1;
--              end if;


            end if;
        end if;
```

On each rising edge we load one of the 64 instructions into our output called inst_out and send it to the IF/ID reg.

IF/ID Register:

The IF/ID register is basically a point that stores the instruction value till the clk increments and allows all stages to proceed, therefore, it is designed with 1 input, 1 output, and a clk input.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity IF_ID_reg is
     port(
         inst_in : in STD_LOGIC_VECTOR(24 downto 0);
         clk : in STD_LOGIC;
         reset : in std_logic;

         inst_out : out STD_LOGIC_VECTOR(24 downto 0)
         );
end IF_ID_reg;
```

Each *rising_edge(clk)* the system will allow an instruction to move into the next stage.

```
architecture behavioral of IF_ID_reg is
begin

    if_id_buffer : process(clk) is
    begin
        if reset = '1' then
            if (rising_edge(clk)) then
                inst_out <= inst_in;
            end if;
        end if;
    end process if_id_buffer;

end behavioral;
```

ID/EX Register:

The ID/EX register holds the values being output from the Register File and holds them for a cycle till the next *rising_edge(clk).* Since the Register File has 4 outputs, we have 4 inputs to store and then output

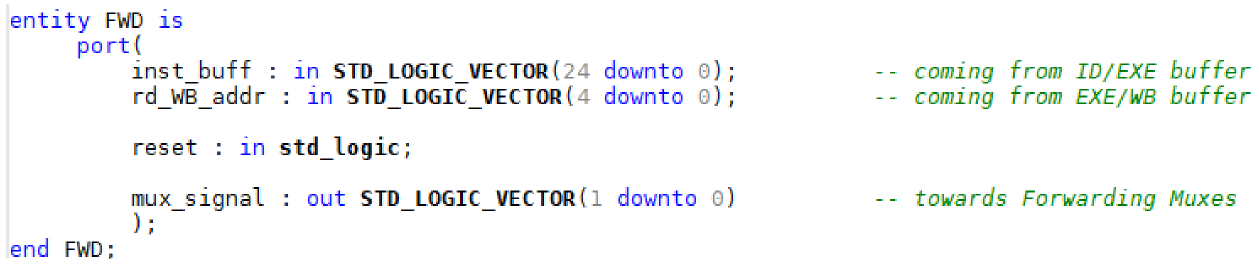
```
entity ID_EX_reg is
     port(
         clk : in std_logic;

         reset : in std_logic;

         inst_in : in STD_LOGIC_VECTOR(24 downto 0);
         val_1 : in STD_LOGIC_VECTOR(127 downto 0);
         val_2 : in STD_LOGIC_VECTOR(127 downto 0);
         val_3 : in STD_LOGIC_VECTOR(127 downto 0);

         inst_out : out STD_LOGIC_VECTOR(24 downto 0);
         val_1_out : out STD_LOGIC_VECTOR(127 downto 0);
         val_2_out : out STD_LOGIC_VECTOR(127 downto 0);
         val_3_out : out STD_LOGIC_VECTOR(127 downto 0)
         );
end ID_EX_reg;
```

```
architecture behavioral of ID_EX_reg is

--  file results_file : text;

begin

    id_ex_buffer : process(clk) is
        variable results_line : line;

    begin
        if reset = '1' then
            if (rising_edge(clk)) then
                inst_out <= inst_in;
                val_1_out <= val_1;
                val_2_out <= val_2;
                val_3_out <= val_3;

--              file_open(results_file, "../results.txt", append_mode);

--              write(results_line, "ID: ");
--              write(results_line, inst_in);
--              writeline(results_file, results_line);

--              write(results_line, inst_in);
--              writeline(results_file, results_line);

            end if;
        end if;
    end process id_ex_buffer;

end behavioral;
```

The instruction is also stored and passed every stage as a means of keeping track of each register address easily at the cost of efficiency.

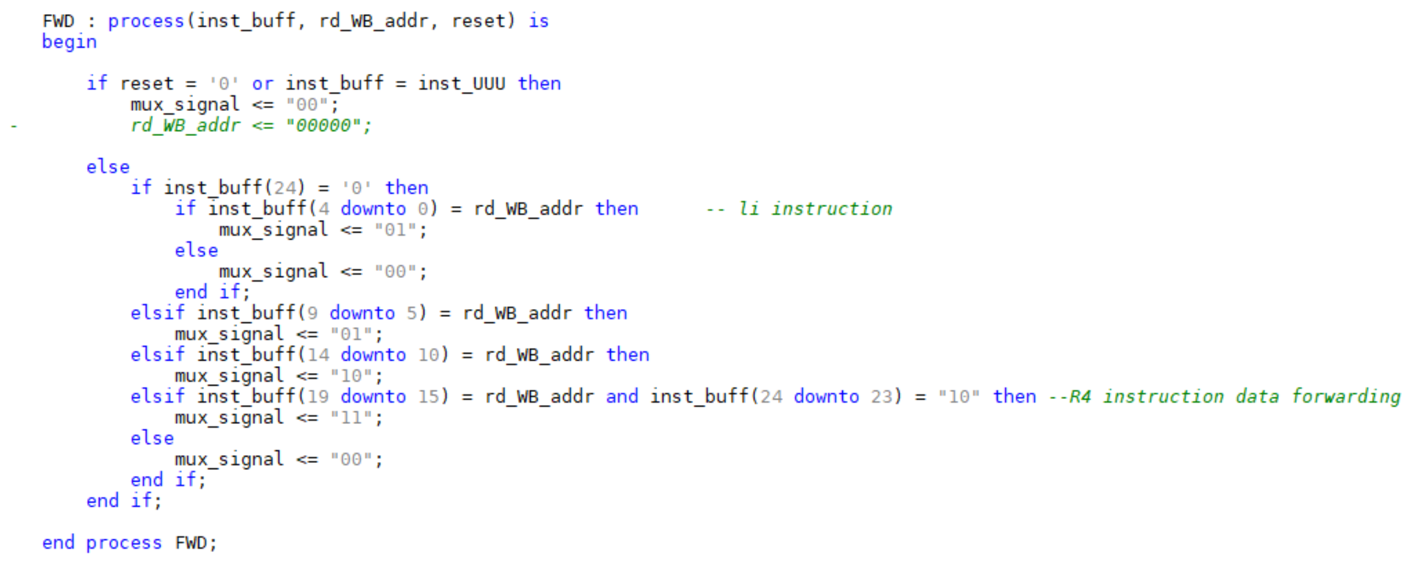
Data Forwarding Unit:

At the Writeback stage, to ensure registers are given their most recent values we send out a writeback address to the data forwarding unit called *rd_WB_addr*.



```
entity FWD is
     port(
         inst_buff : in STD_LOGIC_VECTOR(24 downto 0);           -- coming from ID/EXE buffer
         rd_WB_addr : in STD_LOGIC_VECTOR(4 downto 0);           -- coming from EXE/WB buffer

         reset : in std_logic;

         mux_signal : out STD_LOGIC_VECTOR(1 downto 0)           -- towards Forwarding Muxes
         );
end FWD;
```

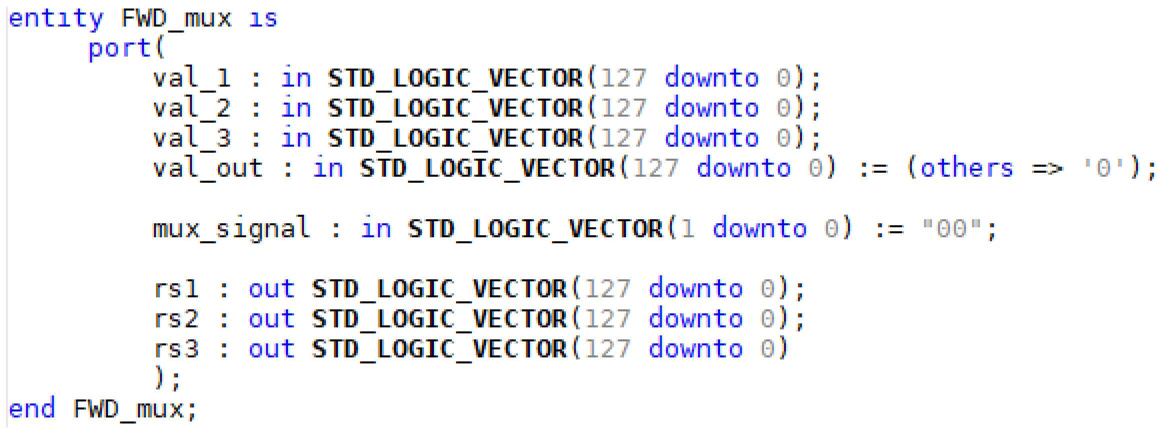
The forwarding unit compares the write back address with the incoming instruction’s addresses to the ALU, these are stored in *inst_buff*. The forwarding unit checks first what type of instruction we are sending to the ALU (R3, R4, load immediate) and then checks rd, rs1, rs2, and rs3 depending on the type of instruction and then sends to the Forwarding Mux a *mux_signal* that tells it whether rd, rs1, rs2, or rs3 is being forwarded.



```
    FWD : process(inst_buff, rd_WB_addr, reset) is
    begin

        if reset = '0' or inst_buff = inst_UUU then
            mux_signal <= "00";
-           rd_WB_addr <= "00000";

        else
            if inst_buff(24) = '0' then
                if inst_buff(4 downto 0) = rd_WB_addr then      -- li instruction
                    mux_signal <= "01";
                else
                    mux_signal <= "00";
                end if;
            elsif inst_buff(9 downto 5) = rd_WB_addr then
                mux_signal <= "01";
            elsif inst_buff(14 downto 10) = rd_WB_addr then
                mux_signal <= "10";
            elsif inst_buff(19 downto 15) = rd_WB_addr and inst_buff(24 downto 23) = "10" then --R4 instruction data forwarding
                mux_signal <= "11";
            else
                mux_signal <= "00";
            end if;
        end if;

    end process FWD;
```

Forwarding Multiplexor:

The Forwarding Mux takes in the 3 outputs from the ID/EX register, 1 output from the Writeback stage called *val_out*, and the *mux_signal* from the forwarding unit to determine whether or not registers need to be replaced by their most recents values in the Writeback stage. If the *mux_signal* is a “01” rs1 needs to be updated, if it’s “10” then rs2 needs to be updated, if the signal is “11” then rs3 gets updated, else the registers are not being written back to. In the event of a Load immediate instruction, rs1 will also hold the value for the rd specified in the load immediate instruction.



```
entity FWD_mux is
    port(
        val_1 : in STD_LOGIC_VECTOR(127 downto 0);
        val_2 : in STD_LOGIC_VECTOR(127 downto 0);
        val_3 : in STD_LOGIC_VECTOR(127 downto 0);
        val_out : in STD_LOGIC_VECTOR(127 downto 0) := (others => '0');

        mux_signal : in STD_LOGIC_VECTOR(1 downto 0) := "00";

        rs1 : out STD_LOGIC_VECTOR(127 downto 0);
        rs2 : out STD_LOGIC_VECTOR(127 downto 0);
        rs3 : out STD_LOGIC_VECTOR(127 downto 0)
        );
end FWD_mux;
```

```
architecture behavioral of FWD_mux is
begin

    FWD_mux : process(mux_signal, val_out) is
    begin

        if mux_signal = "01" then
            rs1 <= val_out;
            rs2 <= val_2;
            rs3 <= val_3;
        elsif mux_signal = "10" then
            rs1 <= val_1;
            rs2 <= val_out;
            rs3 <= val_3;
        elsif mux_signal = "11" then
            rs1 <= val_1;
            rs2 <= val_2;
            rs3 <= val_out;
        else
            rs1 <= val_1;
            rs2 <= val_2;
            rs3 <= val_3;
        end if;

    end process FWD_mux;
```

Multimedia ALU:

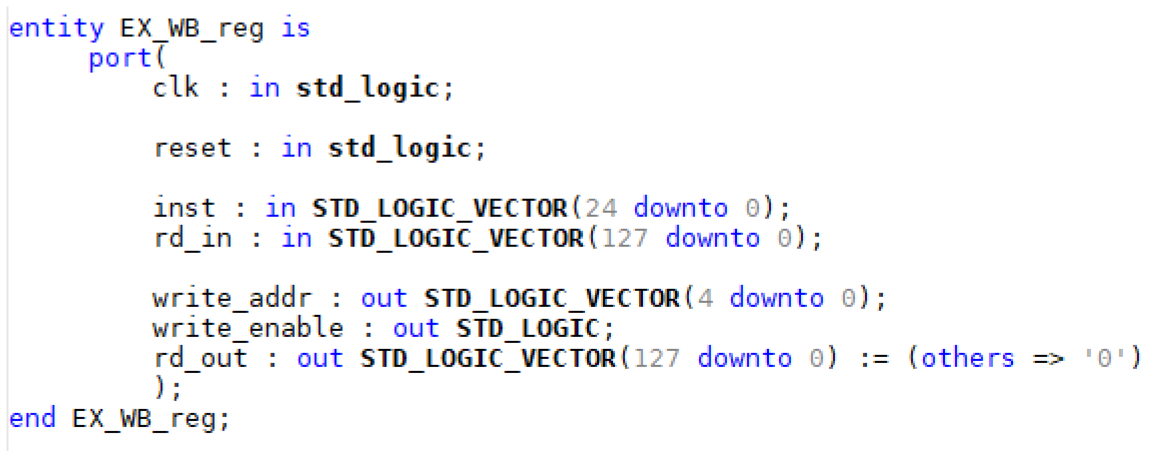
As it was designed in the previous part of the project, the ALU takes in rs1, rs2, and rs3 from the forwarding mux and the instruction is still continuously passed from each stage in the pipeline.

```
entity multimedia_ALU is
     port(
         rs1 : in std_logic_vector(127 downto 0) := (others => '0');
         rs2 : in std_logic_vector(127 downto 0);
         rs3 : in std_logic_vector(127 downto 0);
         inst: in std_logic_vector(24 downto 0);

         rd : out std_logic_vector(127 downto 0)
         );
end multimedia_ALU;
```

After doing internal computations for each specific R3, R4 instructions and load immediate we send the result to rd to which is outputted to the EX/WB register.

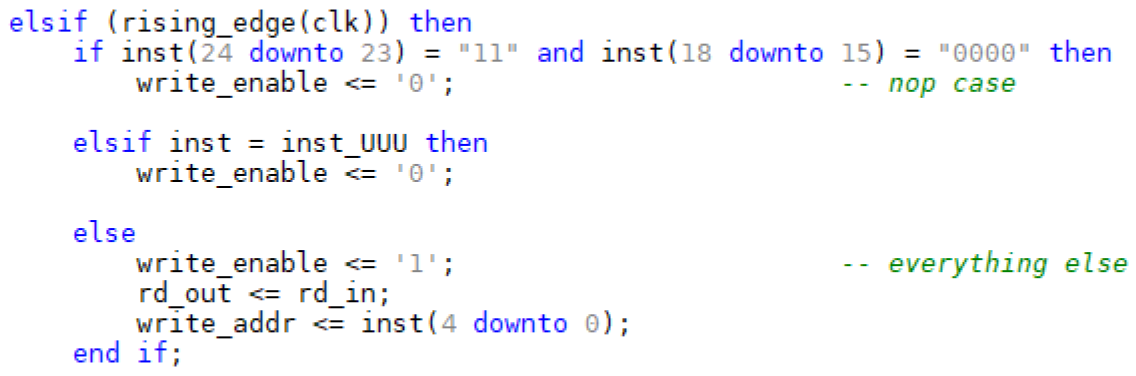
EX/WB Register (Writeback):

In the EX/WB register we pass in the rd from the execute stage and the 25-bit instruction that we’ve been passing each stage.



```vhdl
entity EX_WB_reg is
     port(
          clk : in std_logic;

          reset : in std_logic;

          inst : in STD_LOGIC_VECTOR(24 downto 0);
          rd_in : in STD_LOGIC_VECTOR(127 downto 0);

          write_addr : out STD_LOGIC_VECTOR(4 downto 0);
          write_enable : out STD_LOGIC;
          rd_out : out STD_LOGIC_VECTOR(127 downto 0) := (others => '0')
          );
end EX_WB_reg;
```

We use the 25-bit instruction to determine if we have a nop or not. This is because if we have a "*nop*" we must make sure the *write_enable* is set to 0 otherwise it is always 1 to write back to the Register File and update our registers. Using the last 5 bits of the instruction we can store rd's address in the write_address so we know which register in the register file we must update.

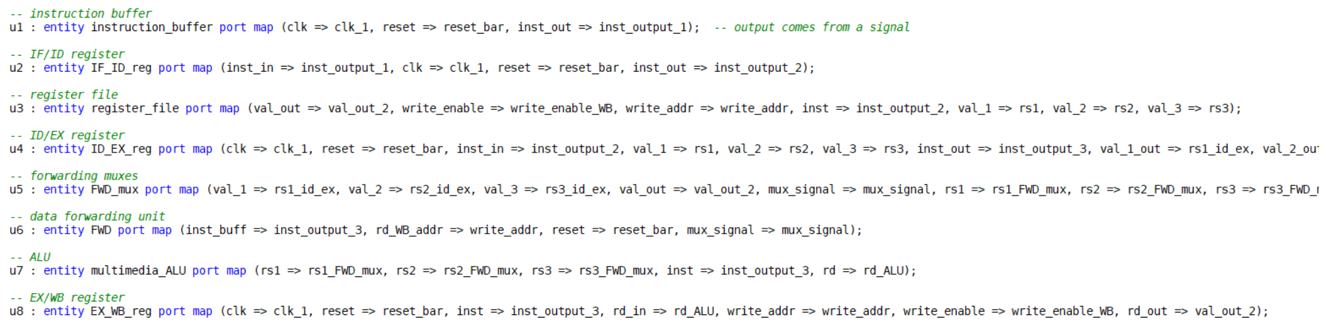


```vhdl
elsif (rising_edge(clk)) then
     if inst(24 downto 23) = "11" and inst(18 downto 15) = "0000" then
          write_enable <= '0';                              -- nop case

     elsif inst = inst_UUU then
          write_enable <= '0';

     else
          write_enable <= '1';                              -- everything else
          rd_out <= rd_in;
          write_addr <= inst(4 downto 0);
     end if;
```

After a rising_edge(clk) signifying the next cycle, we can send all our required write back information to the forwarding unit and Register File. *rd_out* stores the actual value in the register and *write_addr* stores the location of that register in the array located in the Register File.

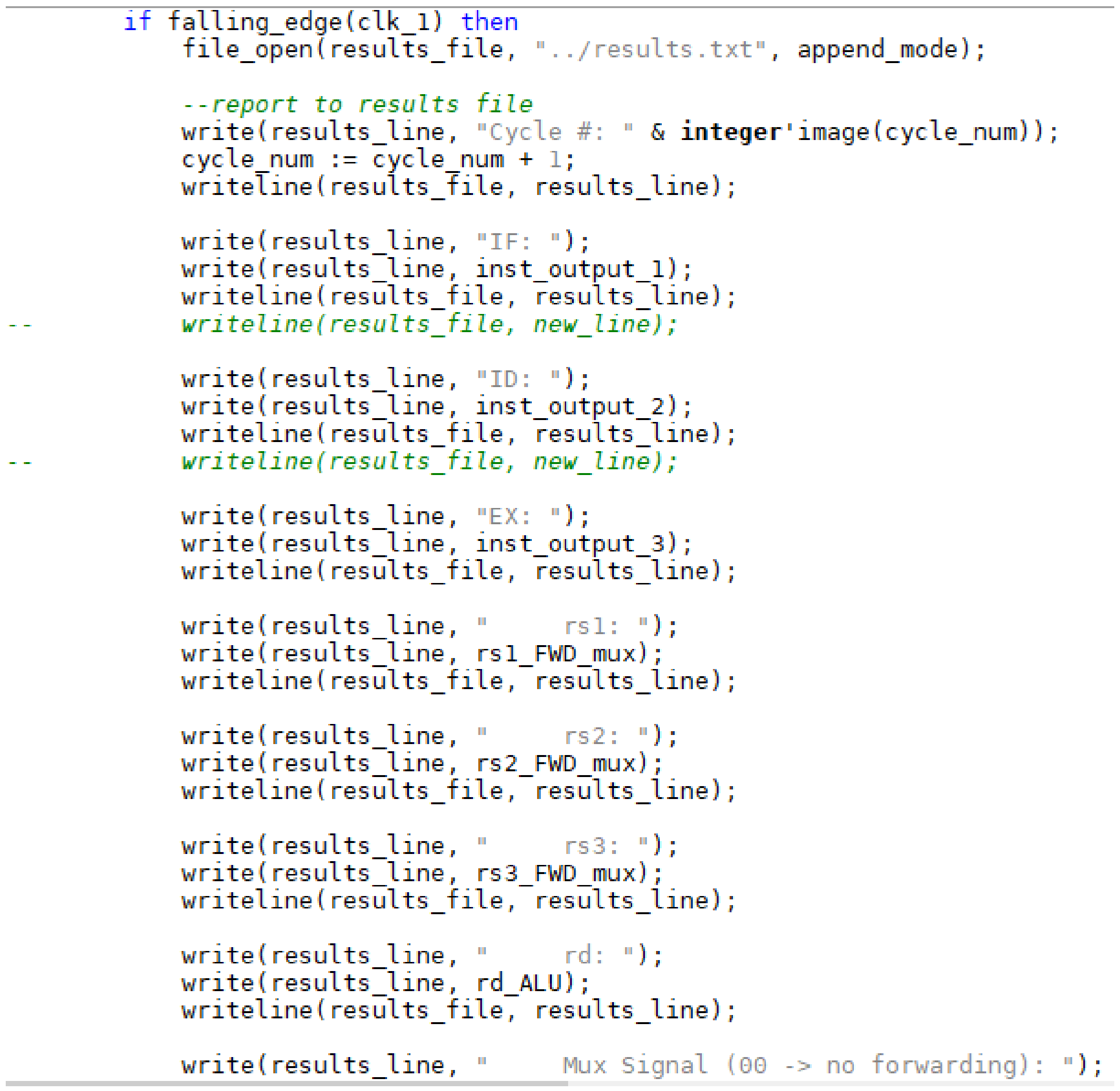
Pipe Processor:

This is our structural architecture for the pipeline that links the system together.



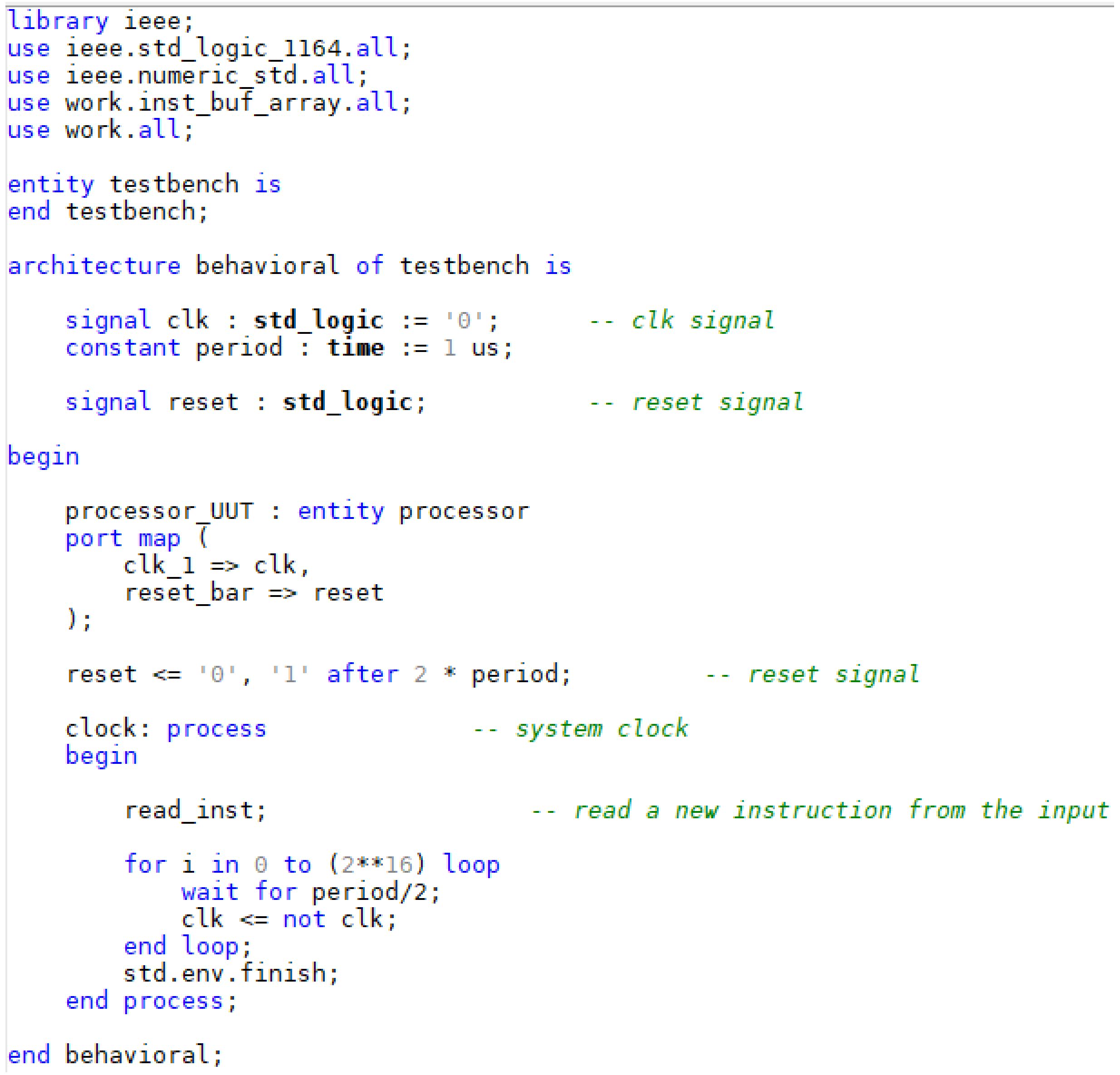
```vhdl
-- instruction buffer
u1 : entity instruction_buffer port map (clk => clk_1, reset => reset_bar, inst_out => inst_output_1);  -- output comes from a signal

-- IF/ID register
u2 : entity IF_ID_reg port map (inst_in => inst_output_1, clk => clk_1, reset => reset_bar, inst_out => inst_output_2);

-- register file
u3 : entity register_file port map (val_out => val_out_2, write_enable => write_enable_WB, write_addr => write_addr, inst => inst_output_2, val_1 => rs1, val_2 => rs2, val_3 => rs3);

-- ID/EX register
u4 : entity ID_EX_reg port map (clk => clk_1, reset => reset_bar, inst_in => inst_output_2, val_1 => rs1, val_2 => rs2, val_3 => rs3, inst_out => inst_output_3, val_1_out => rs1_id_ex, val_2_ou

-- forwarding muxes
u5 : entity FWD_mux port map (val_1 => rs1_id_ex, val_2 => rs2_id_ex, val_3 => rs3_id_ex, val_out => val_out_2, mux_signal => mux_signal, rs1 => rs1_FWD_mux, rs2 => rs2_FWD_mux, rs3 => rs3_FWD_

-- data forwarding unit
u6 : entity FWD port map (inst_buff => inst_output_3, rd_WB_addr => write_addr, reset => reset_bar, mux_signal => mux_signal);

-- ALU
u7 : entity multimedia_ALU port map (rs1 => rs1_FWD_mux, rs2 => rs2_FWD_mux, rs3 => rs3_FWD_mux, inst => inst_output_3, rd => rd_ALU);

-- EX/WB register
u8 : entity EX_WB_reg port map (clk => clk_1, reset => reset_bar, inst => inst_output_3, rd_in => rd_ALU, write_addr => write_addr, write_enable => write_enable_WB, rd_out => val_out_2);
```

This also handles our results file by using the *write()* and *writeline()* functions to list the status of the pipeline for each cycle during the program execution using the "clk" like the stages.


```
        if falling_edge(clk_1) then
            file_open(results_file, "../results.txt", append_mode);

            --report to results file
            write(results_line, "Cycle #: " & integer'image(cycle_num));
            cycle_num := cycle_num + 1;
            writeline(results_file, results_line);

            write(results_line, "IF: ");
            write(results_line, inst_output_1);
            writeline(results_file, results_line);
--          writeline(results_file, new_line);

            write(results_line, "ID: ");
            write(results_line, inst_output_2);
            writeline(results_file, results_line);
--          writeline(results_file, new_line);

            write(results_line, "EX: ");
            write(results_line, inst_output_3);
            writeline(results_file, results_line);

            write(results_line, "     rs1: ");
            write(results_line, rs1_FWD_mux);
            writeline(results_file, results_line);

            write(results_line, "     rs2: ");
            write(results_line, rs2_FWD_mux);
            writeline(results_file, results_line);

            write(results_line, "     rs3: ");
            write(results_line, rs3_FWD_mux);
            writeline(results_file, results_line);

            write(results_line, "     rd: ");
            write(results_line, rd_ALU);
            writeline(results_file, results_line);

            write(results_line, "     Mux Signal (00 -> no forwarding): ");
```

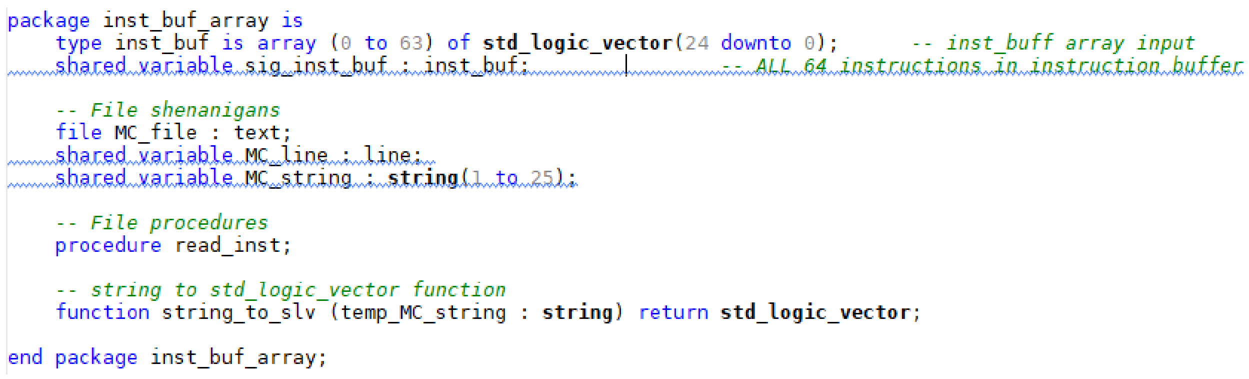
Testbench:

The testbench basically manages the pipeline’s cycles by setting the clock period.



```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.inst_buf_array.all;
use work.all;

entity testbench is
end testbench;

architecture behavioral of testbench is

    signal clk : std_logic := '0';        -- clk signal
    constant period : time := 1 us;

    signal reset : std_logic;             -- reset signal

begin

    processor_UUT : entity processor
    port map (
        clk_1 => clk,
        reset_bar => reset
    );

    reset <= '0', '1' after 2 * period;           -- reset signal

    clock: process                  -- system clock
    begin

        read_inst;                      -- read a new instruction from the input

        for i in 0 to (2**16) loop
            wait for period/2;
            clk <= not clk;
        end loop;
        std.env.finish;
    end process;

end behavioral;
```

The testbench also holds the package filled with the instructions which need to be inputted into the instruction buffer, all 64 25-bit instructions. It gets these instructions from the input.txt which it reads using the functions *read()* and *readline().*



```vhdl
package inst_buf_array is
    type inst_buf is array (0 to 63) of std_logic_vector(24 downto 0);      -- inst_buff array input
    shared variable sig_inst_buf : inst_buf;                   -- ALL 64 instructions in instruction buffer

    -- File shenanigans
    file MC_file : text;
    shared variable MC_line : line;
    shared variable MC_string : string(1 to 25);

    -- File procedures
    procedure read_inst;

    -- string to std_logic_vector function
    function string_to_slv (temp_MC_string : string) return std_logic_vector;

end package inst_buf_array;
```

```
package body inst_buf_array is

    function string_to_slv (temp_MC_string : string) return std_logic_vector is
        variable MC_slv : std_logic_vector(24 downto 0);
    begin
        for i in 1 to 25 loop
            if temp_MC_string(i) = '1' then
                MC_slv(24 - (i-1)) := '1';
            else
                MC_slv(24 - (i-1)) := '0';
            end if;

        end loop;
        return MC_slv;
    end function string_to_slv;

    procedure read_inst is
        variable MC_slv_result : std_logic_vector(24 downto 0);
    begin
        file_open(MC_file, "input.txt", read_mode);

        for i in 0 to 63 loop
            readline(MC_file, MC_line);
            read(MC_line, MC_string);                      -- read line type and convert to string

           MC_slv_result := string_to_slv(MC_string);      -- slv_result from string_to_slv

            sig_inst_buf(i) := MC_slv_result;
        end loop;

        file_close(MC_file);
    end procedure read_inst;

end package body inst_buf_array;
```

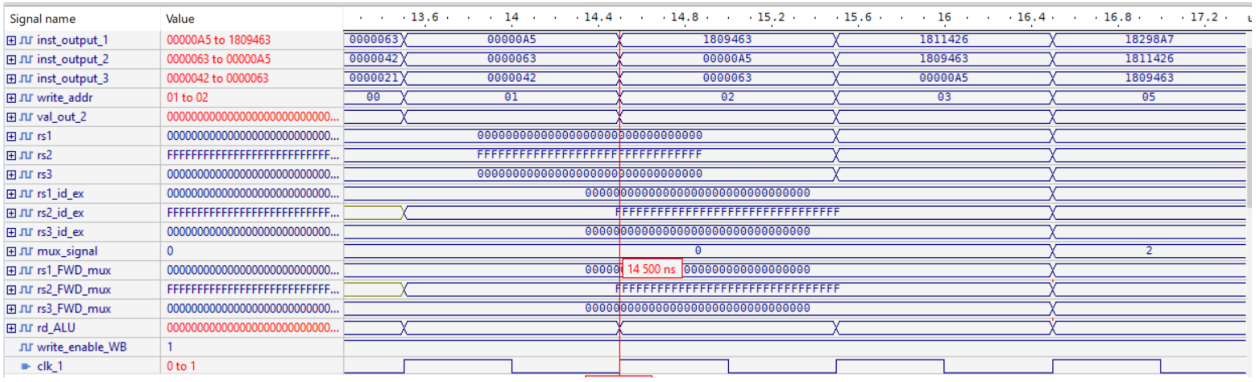
4-Stage Pipeline

Our processor is meant to be a 4-Stage Pipeline processor, where the four stages are represented by Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), and Write Back (WB). As such, our processor can work on 4 different instructions at the same time, but all at different stages. The dataflow in our processor flows from the IF stage, to the ID stage, then to the EX stage, and finally to the WB stage. (IF -> ID -> EX -> WB) As a design rule, the 4 stages represent the components between the 3 buffer registers (IF/ID Register, ID/EX Register, and the EX/WB Register). These registers are meant to store the inputs to components in the next stage, the outputs of the components from the previous stage, and any control signals or instructions that need to pass on from stage to stage. These registers store their values based on a clock signal, while the components between the registers are not dictated by a clock signal.

In the IF stage, the Instruction Buffer component is there to fetch the current instruction and iterate the Program Counter (PC), and to pass the fetched instruction to the IF/ID Register. In the ID stage, the Register File takes in the instruction from the IF/ID Register and returns the relevant registers from the instruction provided (rs3, rs2, rs1, rd), outputs them to the ID/EX Register, and updates any register specified by the “write_enable” and “write_addr” signals from the WB stage. In the EX stage, both execution of the instruction passed in, and data forwarding happens. First, the instruction passed in from the ID/EX Register and the address of the register to be written to in the Register File (“write_addr” from the WB stage) are passed to the Data Forwarding Unit, where its output is a signal to the Forwarding Muxes (“mux_signal”) that decide if data forwarding should occur, and to which mux data forwarding should happen. If the "mux_signal" is "00", then data forwarding does not occur, but if the "mux_signal" was instead "01", "10", or "11", then the values of rs1, rs2, and rs3 respectively should be data-forwarded by the value of the register that is being written to ("val_out"). "Data-forwarded" meaning that the value of "val_out" replaces the value represented in rs1, rs2, or rs3. Even when data forwarding occurs, the Register File would also be updated by the value of "val_out" at the register specified by the register from "write_addr". Once the outputs to the Forwarding muxes are decided, then they are inputted to the ALU along with the instruction, and compute the correct 128-bit value and output it to the EX/WB Register.

Finally, in the WB stage, depending on the instruction (every instruction except "NOP") the "write_enable" signal is set, letting the Register File know that a register is going to be updated. The "write_addr" is also produced to tell the Register File and the Data Forwarding Unit which register is being updated along with the value being computed by the ALU ("val_out") also outputted to both components.

Below is an example from the waveform produced by our simulation showing how 4 instructions are processed in their respective stages:

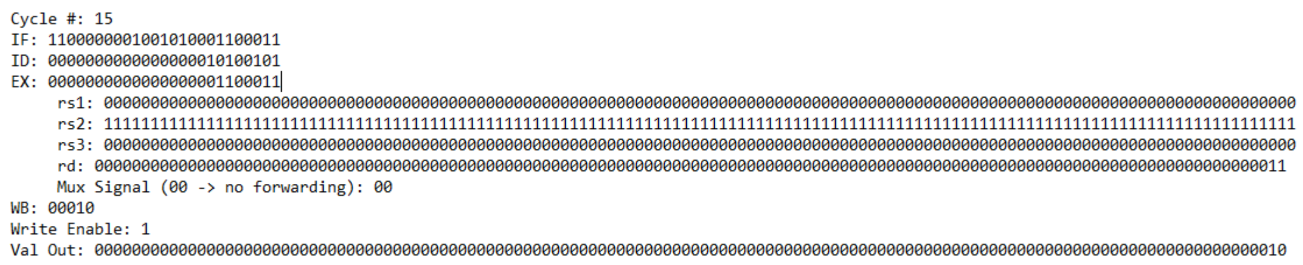


Signal name	Value
inst_output_1	00000A5 to 1809463
inst_output_2	0000063 to 00000A5
inst_output_3	0000042 to 0000063
write_addr	01 to 02
val_out_2	000000000000000000000000000...
rs1	000000000000000000000000000...
rs2	FFFFFFFFFFFFFFFFFFFFFFFFFFF...
rs3	000000000000000000000000000...
rs1_id_ex	000000000000000000000000000...
rs2_id_ex	FFFFFFFFFFFFFFFFFFFFFFFFFFF...
rs3_id_ex	000000000000000000000000000...
mux_signal	0
rs1_FWD_mux	000000000000000000000000000...
rs2_FWD_mux	FFFFFFFFFFFFFFFFFFFFFFFFFFF...
rs3_FWD_mux	000000000000000000000000000...
rd_ALU	000000000000000000000000000...
write_enable_WB	1
clk_1	0 to 1

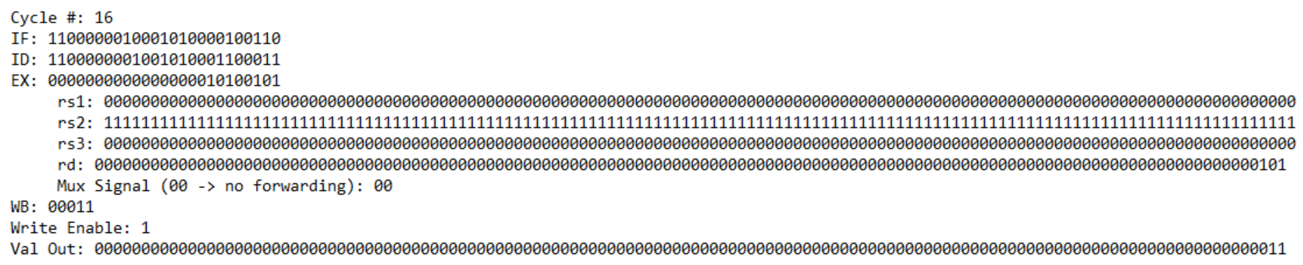
13.6
14
14.4
14.8
15.2
15.6
16
16.4
16.8
17.2
0000063 00000A5 1809463 1811426 18298A7
0000042 0000063 00000A5 1809463 1811426
0000021 0000042 0000063 00000A5 1809463
00 01 02 03 05
2
14 500 ns

By looking at the cycle right after the cursor, we see that our processor is able to process 4 instructions all at the same time but at different stages. The signal "inst_output_1" represents the instruction currently fetched by the Instruction Buffer in the IF stage. "inst_output_2" represents the instruction in the ID stage. "Inst_output_3" represents the instruction in the EX stage, and "write_addr" represents the register that will be written to in the WB stage. Looking at the cycles immediately after the one pointed by the cursor, we see that between each of the "inst_output" instructions and the "write_addr" signals, the instructions from before are passed through to the next stage and the IF stage fetches a new instruction into the pipeline.

Below are the outputs to the "results.txt" file of the two cycles represented by the waveform that are right after where the cursor is pointing:

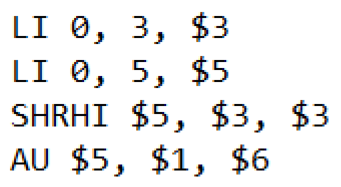


```
Cycle #: 15
IF: 1100000001001010001100011
ID: 0000000000000000010100101
EX: 0000000000000000001100011
     rs1: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
     rs2: 11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
     rs3: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
     rd: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011
     Mux Signal (00 -> no forwarding): 00
WB: 00010
Write Enable: 1
Val Out: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010
```



```
Cycle #: 16
IF: 1100000010001010000100110
ID: 1100000001001010001100011
EX: 0000000000000000010100101
     rs1: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
     rs2: 11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
     rs3: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
     rd: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000101
     Mux Signal (00 -> no forwarding): 00
WB: 00011
Write Enable: 1
Val Out: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011
```

Like with the waveform, the instructions from one stage pass to the next every cycle. Similarly to “inst_output_1”, “inst_output_2” and “inst_output_3”, the “IF”, “ID”, and “EX” sections show the instructions in those stages, and “WB” represents the “write_addr” from the waveform. The instructions themselves can be found below from the input into our assembler:



```
LI 0, 3, $3
LI 0, 5, $5
SHRHI $5, $3, $3
AU $5, $1, $6
```

With the following instructions, the instruction “LI 0, 3, $3” represents a Load Immediate instruction with the binary machine code of “0000000000000000001100011” (as seen from the instruction in the “EX” stage in Cycle #15 from the Results File) and with the Hexadecimal representation of “0000063” (as seen from the “inst_output_3” signal from the cycle pointed by the cursor from the waveform). The instruction “AU $5, $1, $6” represents the Add Word Unsigned instruction with the binary machine code of “1100000010001010000100110” (as seen from the instruction in the “IF” stage in Cycle #16 from the Results File) and with the Hexadecimal notation of “1811426” (as seen from the “inst_output_1” signal from the cycle right after the cycle pointed by the cursor from the waveform).

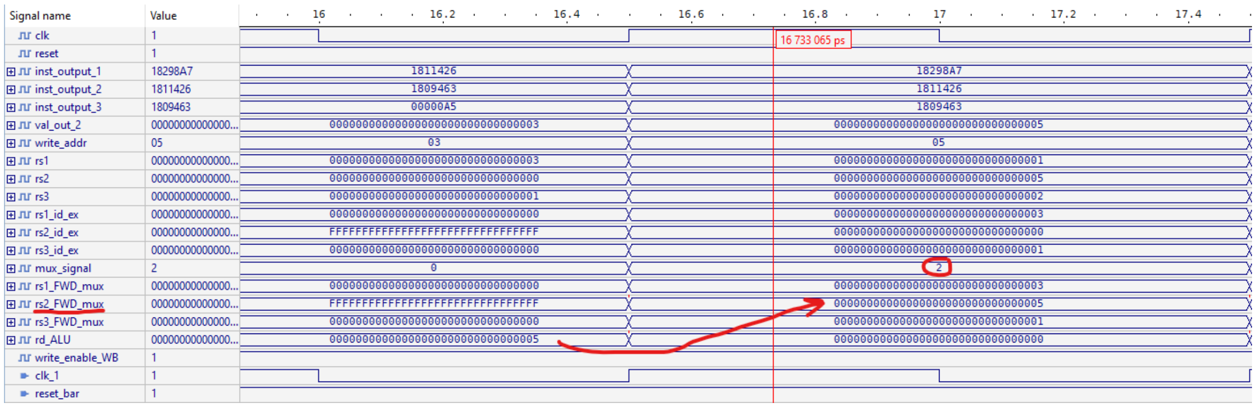
Data Forwarding

Here is an example of our Data Forwarding Unit functioning. Here we can see that 5 is loaded to register $5 and in the next cycle the next instruction requires $5 as rs2, luckily our data forwarding replaces $5 with the updated value of 5 which allows the next instruction to properly function with no data hazards.

(LI load index, 16-bit immediate, rd)

(Instruction rs2, rs1, rd)

```
LI 0, 5, $5
SHRHI $5, $3, $3
```



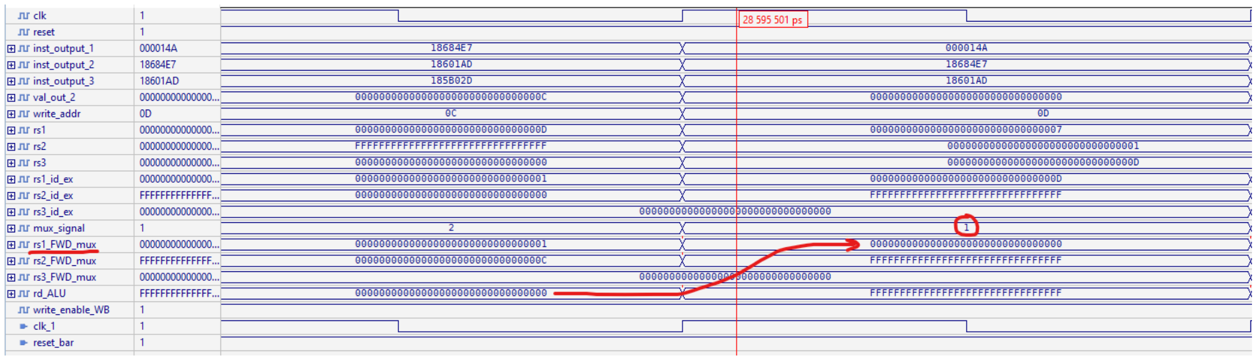
Signal name	Value		
clk	1		
reset	1		
inst_output_1	18298A7	1811426	18298A7
inst_output_2	1811426	1809463	1811426
inst_output_3	1809463	00000A5	1809463
val_out_2	00000000000000...	00000000000000000000000000000003	00000000000000000000000000000005
write_addr	05	03	05
rs1	00000000000000...	00000000000000000000000000000003	00000000000000000000000000000001
rs2	00000000000000...	00000000000000000000000000000000	00000000000000000000000000000005
rs3	00000000000000...	00000000000000000000000000000001	00000000000000000000000000000002
rs1_id_ex	00000000000000...	00000000000000000000000000000000	00000000000000000000000000000003
rs2_id_ex	00000000000000...	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	00000000000000000000000000000000
rs3_id_ex	00000000000000...	00000000000000000000000000000000	00000000000000000000000000000001
mux_signal	2	0	2
rs1_FWD_mux	00000000000000...	00000000000000000000000000000000	00000000000000000000000000000003
rs2_FWD_mux	00000000000000...	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	00000000000000000000000000000005
rs3_FWD_mux	00000000000000...	00000000000000000000000000000000	00000000000000000000000000000001
rd_ALU	00000000000000...	00000000000000000000000000000005	00000000000000000000000000000000
write_enable_WB	1		
clk_1	1		
reset_bar	1		

Here is another example of Data Forwarding where the first instruction updates $13 with all zeros. The next instruction requires $13 as its rs1 and it so we forward that data and replace it with its most recent value.

(Instruction rs2, rs1, rd)

(Instruction rs2, rs1, rd)

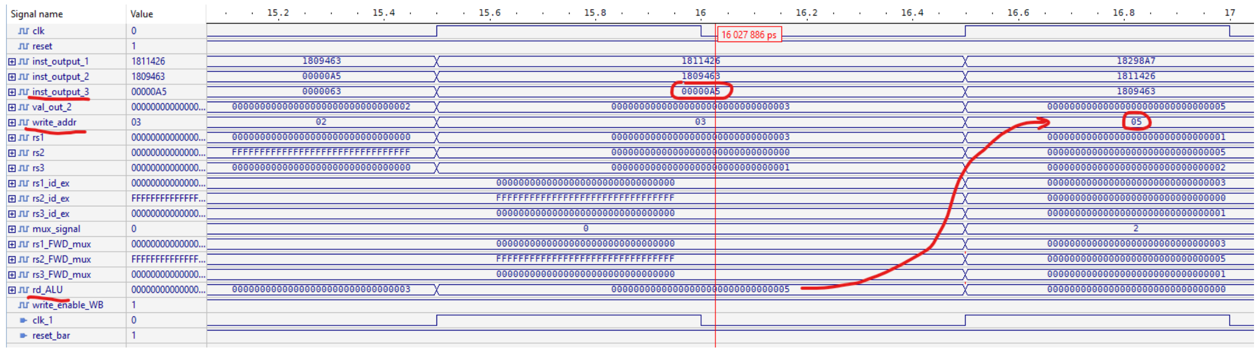
```
AND $12, $1, $13
INVB $0, $13, $13
```



clk	1		
reset	1		
inst_output_1	000014A	18684E7	000014A
inst_output_2	18684E7	18601AD	18684E7
inst_output_3	18601AD	185B02D	18601AD
val_out_2	00000000000000...	0000000000000000000000000000000C	00000000000000000000000000000000
write_addr	0D	0C	0D
rs1	00000000000000...	0000000000000000000000000000000D	00000000000000000000000000000007
rs2	00000000000000...	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	00000000000000000000000000000001
rs3	00000000000000...	00000000000000000000000000000000	0000000000000000000000000000000D
rs1_id_ex	00000000000000...	00000000000000000000000000000001	0000000000000000000000000000000D
rs2_id_ex	FFFFFFFFFFFFFF...	00000000000000000000000000000000	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
rs3_id_ex	00000000000000...	00000000000000000000000000000000	
mux_signal	1	2	1
rs1_FWD_mux	00000000000000...	00000000000000000000000000000001	00000000000000000000000000000000
rs2_FWD_mux	FFFFFFFFFFFFFF...	0000000000000000000000000000000C	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
rs3_FWD_mux	00000000000000...	00000000000000000000000000000000	
rd_ALU	FFFFFFFFFFFFFF...	00000000000000000000000000000000	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
write_enable_WB	1		
clk_1	1		
reset_bar	1		

Instruction Execution

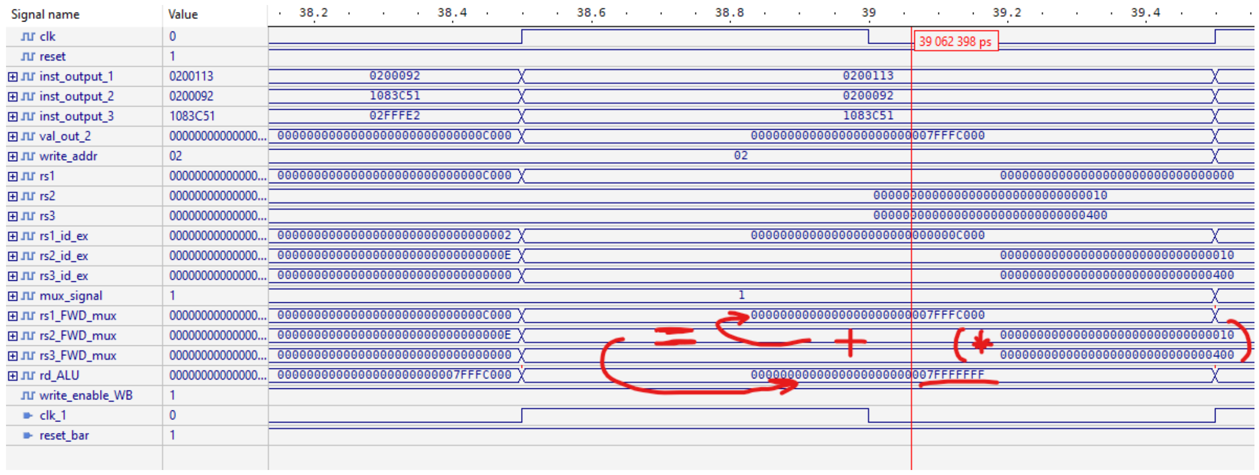
LI



Signal name	Value
clk	0
reset	1
inst_output_1	1811426
inst_output_2	1809463
inst_output_3	00000A5
val_out_2	0000000000000...
write_addr	03
rs1	0000000000000...
rs2	0000000000000...
rs3	0000000000000...
rs1_id_ex	0000000000000...
rs2_id_ex	FFFFFFFFFFFFF...
rs3_id_ex	0000000000000...
mux_signal	0
rs1_FWD_mux	0000000000000...
rs2_FWD_mux	FFFFFFFFFFFFF...
rs3_FWD_mux	0000000000000...
rd_ALU	0000000000000...
write_enable_WB	1
clk_1	0
reset_bar	1

Here we load $5 as shown by *write_addr* with a value of 5 as shown by *rd_ALU* and the instruction is 00000A5 in hex which is equivalent to LI 0, 5, $5 **(LI load index, 16-bit immediate, rd)** in our own MIPs syntax.

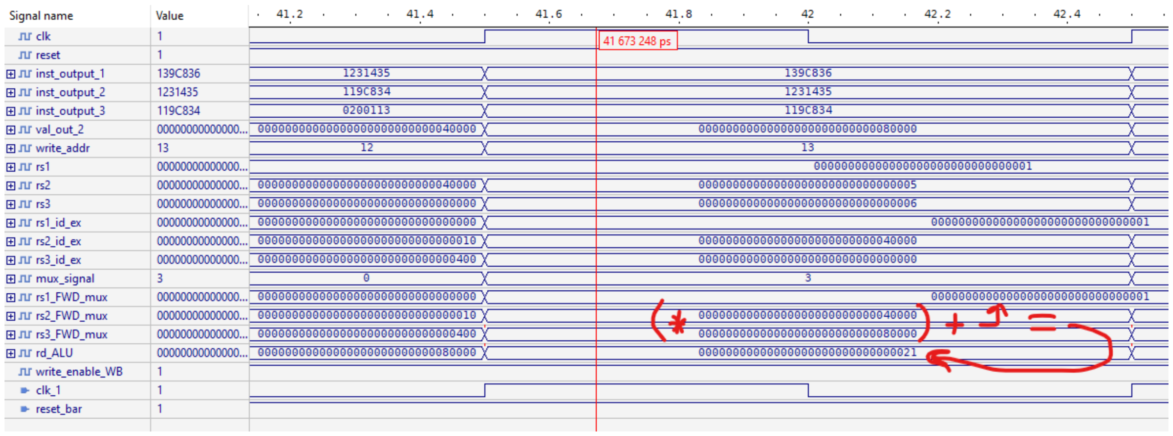
Signed Integer Multiply-Add Low with Saturation



Signal name	Value
clk	0
reset	1
inst_output_1	0200113
inst_output_2	0200092
inst_output_3	1083C51
val_out_2	0000000000000...
write_addr	02
rs1	0000000000000...
rs2	0000000000000...
rs3	0000000000000...
rs1_id_ex	0000000000000...
rs2_id_ex	0000000000000...
rs3_id_ex	0000000000000...
mux_signal	1
rs1_FWD_mux	0000000000000...
rs2_FWD_mux	0000000000000...
rs3_FWD_mux	0000000000000...
rd_ALU	0000000000000...
write_enable_WB	1
clk_1	0
reset_bar	1

Multiply the low 16 bits of rs3 by the low 16 bits of rs2 and then add rs1 then check for overflow and saturation. In our test we used [31:0] to test our overflow case, by doing 1024(rs3) * 16(rs2) + (2^17) we get overflow and as you can see we get 7FFFFFFF in rd which is the max value we can represent in 32 bits. **(Other cases were shown in the previously submitted report)**

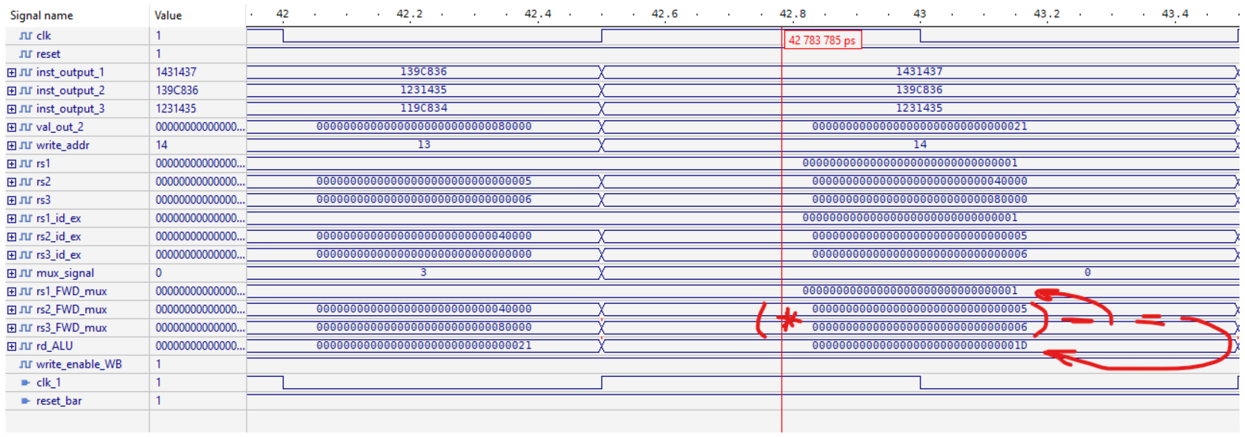
Signed Integer Multiply-Add High with Saturation



Signal name	Value
clk	1
reset	1
inst_output_1	139C836
inst_output_2	1231435
inst_output_3	119C834
val_out_2	00000000000000...
write_addr	13
rs1	00000000000000...
rs2	00000000000000...
rs3	00000000000000...
rs1_id_ex	00000000000000...
rs2_id_ex	00000000000000...
rs3_id_ex	00000000000000...
mux_signal	3
rs1_FWD_mux	00000000000000...
rs2_FWD_mux	00000000000000...
rs3_FWD_mux	00000000000000...
rd_ALU	00000000000000...
write_enable_WB	1
clk_1	1
reset_bar	1

Multiply the high 16 bits of rs3 by the high 16 bits of rs2 and then add rs1 then check for overflow and saturation. In our test we used [31:0] to do 8 * 4 + 1 = 33 which is 21 in hex.

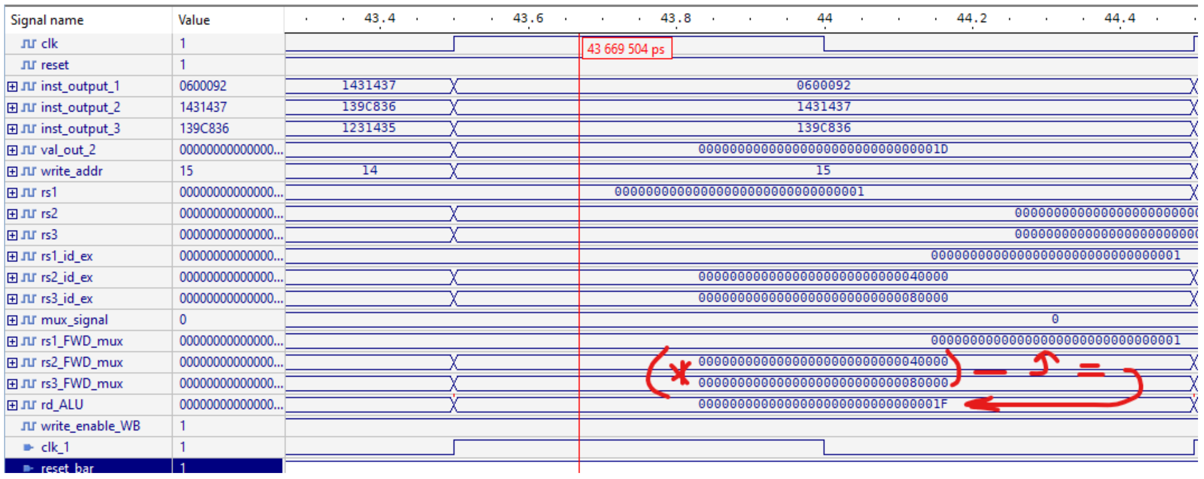
Signed Integer Multiply-Subtract Low with Saturation



Signal name	Value
clk	1
reset	1
inst_output_1	1431437
inst_output_2	139C836
inst_output_3	1231435
val_out_2	00000000000000...
write_addr	14
rs1	00000000000000...
rs2	00000000000000...
rs3	00000000000000...
rs1_id_ex	00000000000000...
rs2_id_ex	00000000000000...
rs3_id_ex	00000000000000...
mux_signal	0
rs1_FWD_mux	00000000000000...
rs2_FWD_mux	00000000000000...
rs3_FWD_mux	00000000000000...
rd_ALU	00000000000000...
write_enable_WB	1
clk_1	1
reset_bar	1

Multiply the low 16 bits of rs3 by the low 16 bits of rs2 and then subtract rs1 then check for overflow and saturation. In our test we used [31:0] to do 6 * 5 - 1 or 1D in hex.

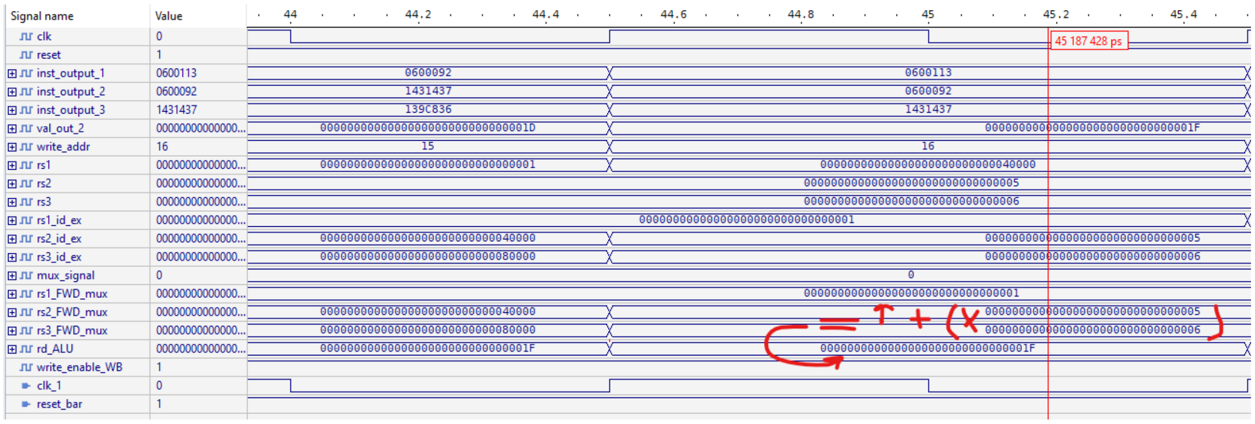
Signed Integer Multiply-Subtract High with Saturation



Signal name	Value
clk	1
reset	1
inst_output_1	0600092
inst_output_2	1431437
inst_output_3	139C836
val_out_2	00000000000000...
write_addr	15
rs1	00000000000000...
rs2	00000000000000...
rs3	00000000000000...
rs1_id_ex	00000000000000...
rs2_id_ex	00000000000000...
rs3_id_ex	00000000000000...
mux_signal	0
rs1_FWD_mux	00000000000000...
rs2_FWD_mux	00000000000000...
rs3_FWD_mux	00000000000000...
rd_ALU	00000000000000...
write_enable_WB	1
clk_1	1
reset_bar	1

Multiply the high 16 bits of rs3 by the high 16 bits of rs2 and then subtract rs1 then check for overflow and saturation. In our test we used [31:0] to show 8 * 4 -1 = 31 or 1F in hex.

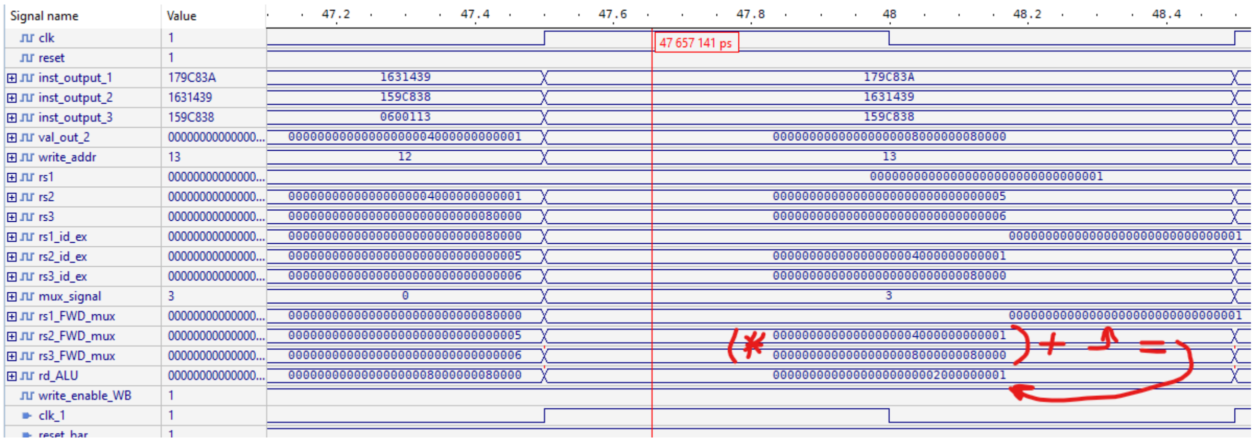
Signed Long Integer Multiply-Add Low with Saturation



Signal name	Value
clk	0
reset	1
inst_output_1	0600113
inst_output_2	0600092
inst_output_3	1431437
val_out_2	00000000000000...
write_addr	16
rs1	00000000000000...
rs2	00000000000000...
rs3	00000000000000...
rs1_id_ex	00000000000000...
rs2_id_ex	00000000000000...
rs3_id_ex	00000000000000...
mux_signal	0
rs1_FWD_mux	00000000000000...
rs2_FWD_mux	00000000000000...
rs3_FWD_mux	00000000000000...
rd_ALU	00000000000000...
write_enable_WB	1
clk_1	0
reset_bar	1

Multiply the low 32 bits of rs3 by the low 32 bits of rs2 and then add rs1 then check for overflow and saturation. In our test we used [63:0] to show 6 * 5 + 1 = 31 or 1F in hex.

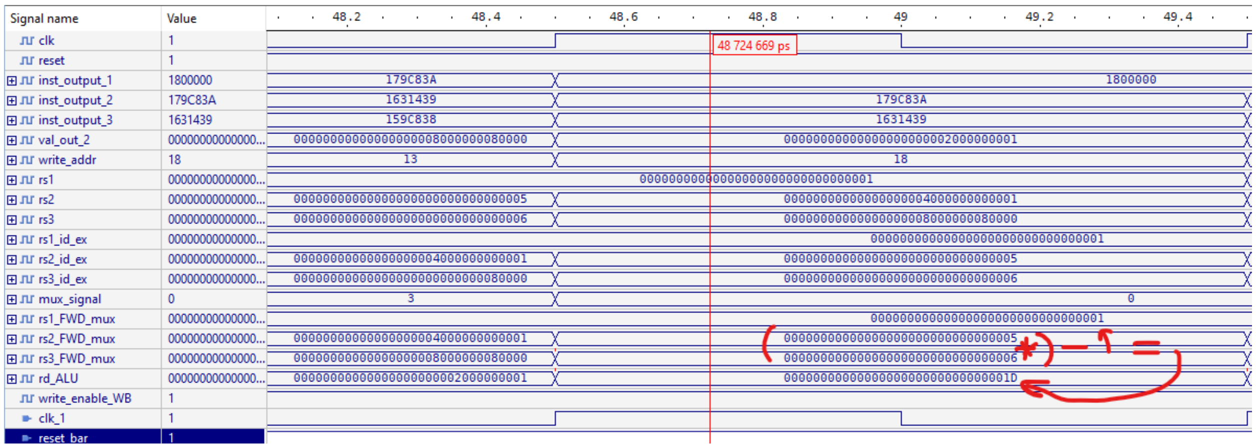
Signed Long Integer Multiply-Add High with Saturation



Signal name	Value
clk	1
reset	1
inst_output_1	179C83A
inst_output_2	1631439
inst_output_3	159C838
val_out_2	00000000000000...
write_addr	13
rs1	00000000000000...
rs2	00000000000000...
rs3	00000000000000...
rs1_id_ex	00000000000000...
rs2_id_ex	00000000000000...
rs3_id_ex	00000000000000...
mux_signal	3
rs1_FWD_mux	00000000000000...
rs2_FWD_mux	00000000000000...
rs3_FWD_mux	00000000000000...
rd_ALU	00000000000000...
write_enable_WB	1
clk_1	1
reset_bar	1

Multiply the high 32 bits of rs3 by the high 32 bits of rs2 and then add rs1 then check for overflow and saturation. In our test we used [63:0] to do 524288 * 262144 + 1 = 137438953473.

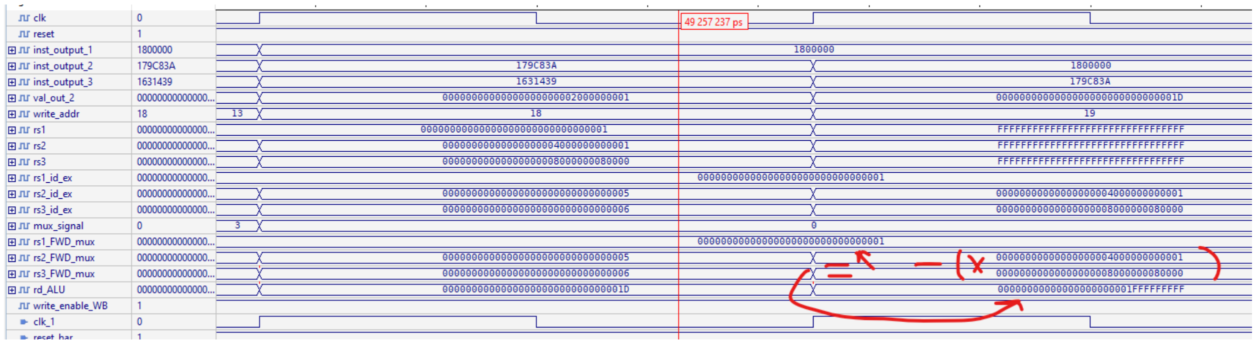
Signed Long Integer Multiply-Subtract Low with Saturation



Signal name	Value
clk	1
reset	1
inst_output_1	1800000
inst_output_2	179C83A
inst_output_3	1631439
val_out_2	00000000000000...
write_addr	18
rs1	00000000000000...
rs2	00000000000000...
rs3	00000000000000...
rs1_id_ex	00000000000000...
rs2_id_ex	00000000000000...
rs3_id_ex	00000000000000...
mux_signal	0
rs1_FWD_mux	00000000000000...
rs2_FWD_mux	00000000000000...
rs3_FWD_mux	00000000000000...
rd_ALU	00000000000000...
write_enable_WB	1
clk_1	1
reset_bar	1

Multiply the low 32 bits of rs3 by the low 32 bits of rs2 and then subtract rs1 then check for overflow and saturation. In our test we used [63:0] to do 6 * 5 - 1 = 29 or 1D in hex.

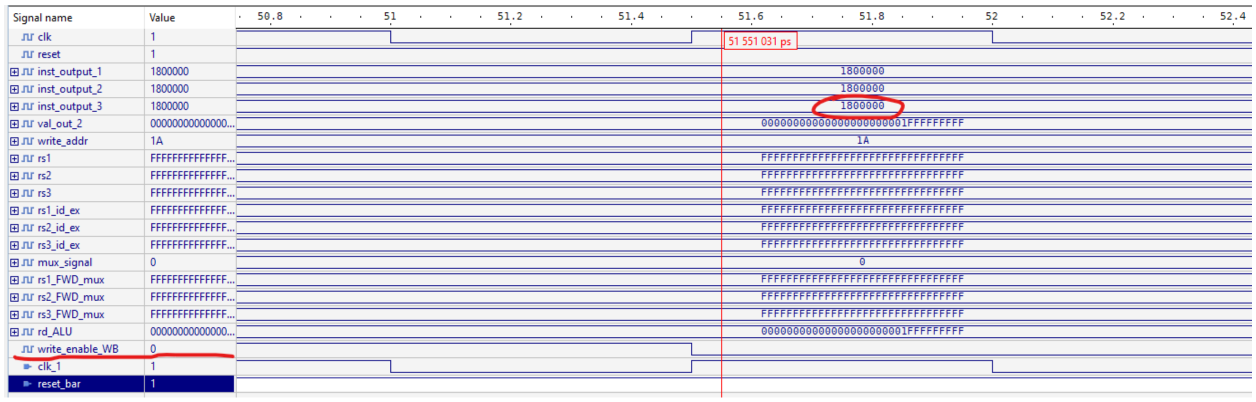
Signed Long Integer Multiply-Subtract High with Saturation



clk	0
reset	1
inst_output_1	1800000
inst_output_2	179C83A
inst_output_3	1631439
val_out_2	00000000000000...
write_addr	18
rs1	00000000000000...
rs2	00000000000000...
rs3	00000000000000...
rs1_id_ex	00000000000000...
rs2_id_ex	00000000000000...
rs3_id_ex	00000000000000...
mux_signal	0
rs1_FWD_mux	00000000000000...
rs2_FWD_mux	00000000000000...
rs3_FWD_mux	00000000000000...
rd_ALU	00000000000000...
write_enable_WB	1
clk_1	0
reset_bar	1

Multiply the high 32 bits of rs3 by the high 32 bits of rs2 and then subtract rs1 then check for overflow and saturation. In our test we used [63:0] to do 524288 * 262144 - 1 = 137438953471.

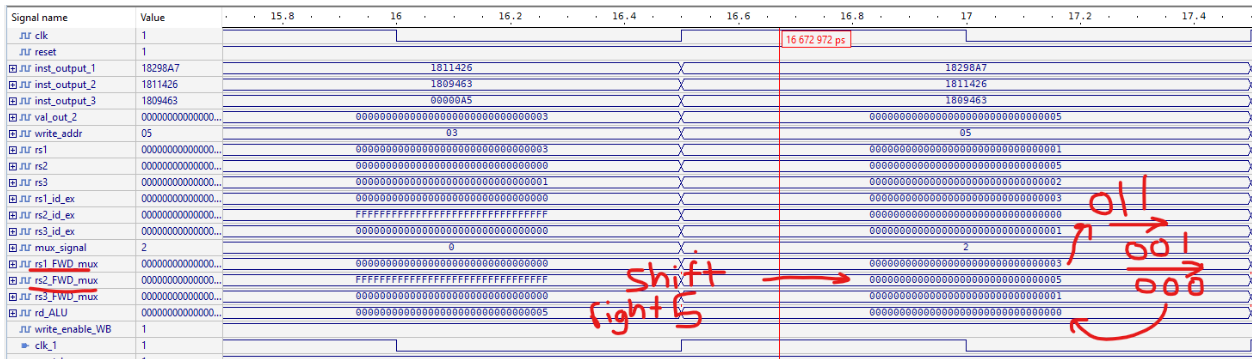
NOP



Signal name	Value
clk	1
reset	1
inst_output_1	1800000
inst_output_2	1800000
inst_output_3	1800000
val_out_2	00000000000000...
write_addr	1A
rs1	FFFFFFFFFFFFF...
rs2	FFFFFFFFFFFFF...
rs3	FFFFFFFFFFFFF...
rs1_id_ex	FFFFFFFFFFFFF...
rs2_id_ex	FFFFFFFFFFFFF...
rs3_id_ex	FFFFFFFFFFFFF...
mux_signal	0
rs1_FWD_mux	FFFFFFFFFFFFF...
rs2_FWD_mux	FFFFFFFFFFFFF...
rs3_FWD_mux	FFFFFFFFFFFFF...
rd_ALU	00000000000000...
write_enable_WB	0
clk_1	1
reset_bar	1

With NOP (inst = 1800000), after taking in values for rs1, rs2, and rs3, the value of rd does not change at all, just as the NOP instruction is supposed to do and our write_enable = 0 so the register file is not tampered with.

SHRHI



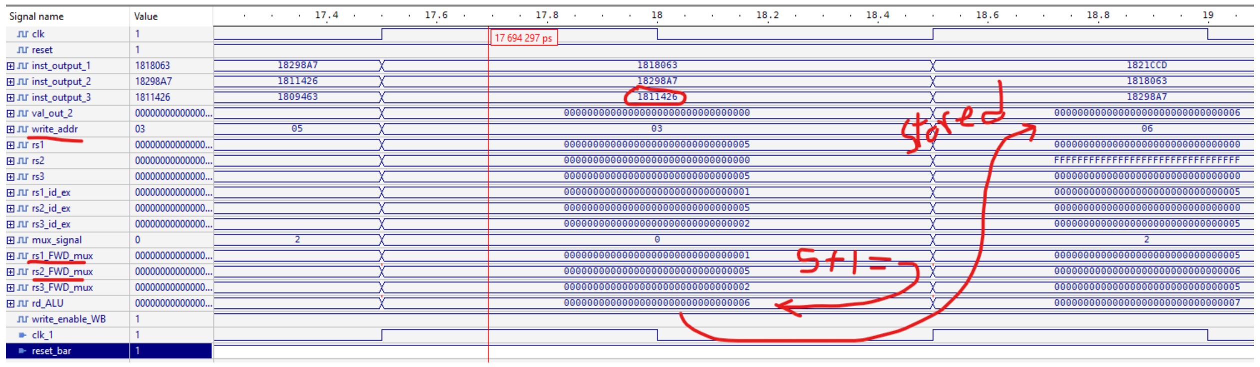
```
Signal name      Value            15.8   16   16.2   16.4   |   16.6   16.8   17   17.2   17.4
clk              1
reset            1
inst_output_1    18298A7          1811426                          | 18298A7
inst_output_2    1811426          1809463                          | 1811426
inst_output_3    1809463          00000A5                          | 1809463
val_out_2        00000000000000…  00000000000000000000000000000003 | 00000000000000000000000000000005
write_addr       05               03                               | 05
rs1              00000000000000…  00000000000000000000000000000003 | 00000000000000000000000000000001
rs2              00000000000000…  00000000000000000000000000000000 | 00000000000000000000000000000005
rs3              00000000000000…  00000000000000000000000000000001 | 00000000000000000000000000000002
rs1_id_ex        00000000000000…  00000000000000000000000000000000 | 00000000000000000000000000000003
rs2_id_ex        00000000000000…  FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF | 00000000000000000000000000000000
rs3_id_ex        00000000000000…  00000000000000000000000000000000 | 00000000000000000000000000000001
mux_signal       2                0                                | 2
rs1_FWD_mux      00000000000000…  00000000000000000000000000000000 | 00000000000000000000000000000003
rs2_FWD_mux      00000000000000…  FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF | 00000000000000000000000000000005
rs3_FWD_mux      00000000000000…  00000000000000000000000000000000 | 00000000000000000000000000000001
rd_ALU           00000000000000…  00000000000000000000000000000005 | 00000000000000000000000000000000
write_enable_WB  1
clk_1            1
```

With SHRHI, (inst = 1809463),
(Instruction rs2, rs1, rd)

```
SHRHI $5, $3, $3
```

Register $5 stores a value of 5 and register $3 stores a value of 3. Here we are shifting $3 over 5 to the right which then creates a value of 0 which is then stored back into $3.

AU



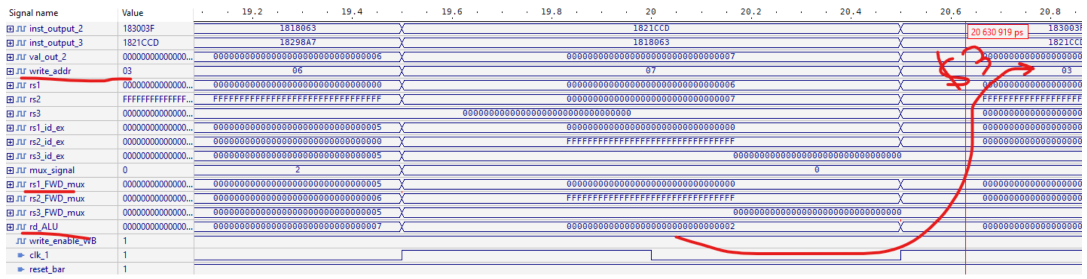
```
Signal name      Value            17.4   |   17.6   17.8   18   18.2   18.4   |   18.6   18.8   19
clk              1
reset            1
inst_output_1    1818063          18298A7 | 1818063                          | 1821CCD
inst_output_2    18298A7          1811426 | 18298A7                          | 1818063
inst_output_3    1811426          1809463 | 1811426                          | 18298A7
val_out_2        00000000000000…          | 00000000000000000000000000000000 | 00000000000000000000000000000006
write_addr       03               05      | 03                               | 06
rs1              00000000000000…          | 00000000000000000000000000000005 | 00000000000000000000000000000000
rs2              00000000000000…          | 00000000000000000000000000000000 | FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
rs3              00000000000000…          | 00000000000000000000000000000005 | 00000000000000000000000000000000
rs1_id_ex        00000000000000…          | 00000000000000000000000000000001 | 00000000000000000000000000000005
rs2_id_ex        00000000000000…          | 00000000000000000000000000000005 | 00000000000000000000000000000000
rs3_id_ex        00000000000000…          | 00000000000000000000000000000002 | 00000000000000000000000000000005
mux_signal       0                2       | 0                                | 2
rs1_FWD_mux      00000000000000…          | 00000000000000000000000000000001 | 00000000000000000000000000000005
rs2_FWD_mux      00000000000000…          | 00000000000000000000000000000005 | 00000000000000000000000000000006
rs3_FWD_mux      00000000000000…          | 00000000000000000000000000000002 | 00000000000000000000000000000005
rd_ALU           00000000000000…          | 00000000000000000000000000000006 | 00000000000000000000000000000007
write_enable_WB  1
clk_1            1
reset_bar        1
```

With AU, (inst = 1811426),
(Instruction rs2, rs1, rd)

```
AU $5, $1, $6
```

The packed 32-bit words in $5 and $1 have the values 0, 0, 0, 5 and 0, 0, 0, 1 (from the most significant words to the least significant) respectively. After AU, the correct result of 0, 0, 0, 6 in the packed 32-bit words of $6 are correctly computed.

CNT1H



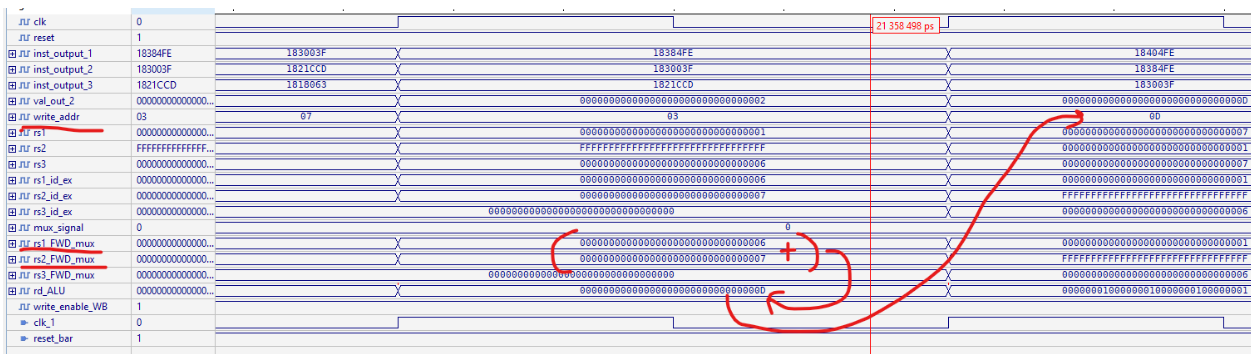
With CNT1H, (inst =1818063),

(Instruction rs2, rs1, rd)

`CNT1H $0, $3, $3`

The counts of ‘1’ in each halfword in $3 are computed back into $3, as the values of 0, 0, 0, 0, 0, 0, 0, and 3 (from the most significant halfwords to the least significant) are represented in the corresponding halfwords of rd which is $3 as well, with the counts of 0, 0, 0, 0, 0, 0, 0, and 2 respectively.

AHS



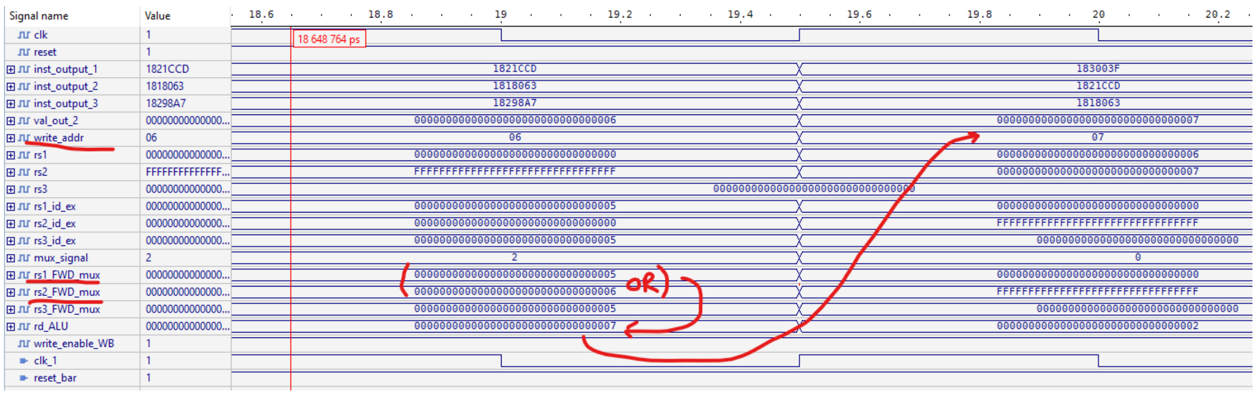
With AHS, (inst = 1821CCD),

(Instruction rs2, rs1, rd)

`AHS $7, $6, $13`

We test the least significant 16-bits to do 7 + 6 using registers $7 and $6 which store those values and get 13 which is then stored into $13.

OR



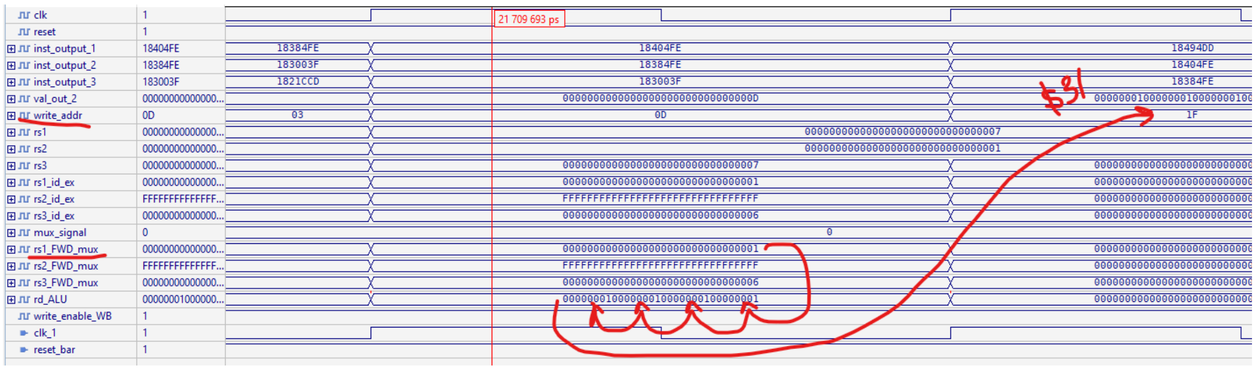
With OR, (inst = 18298A7),

(Instruction rs2, rs1, rd)

```
OR $6, $5, $7
```

The logical OR of $6 which stores 6 (0110) and $5 which stores 5 (0101) gets us 7 (0111) and it's stored in $7.

BCW



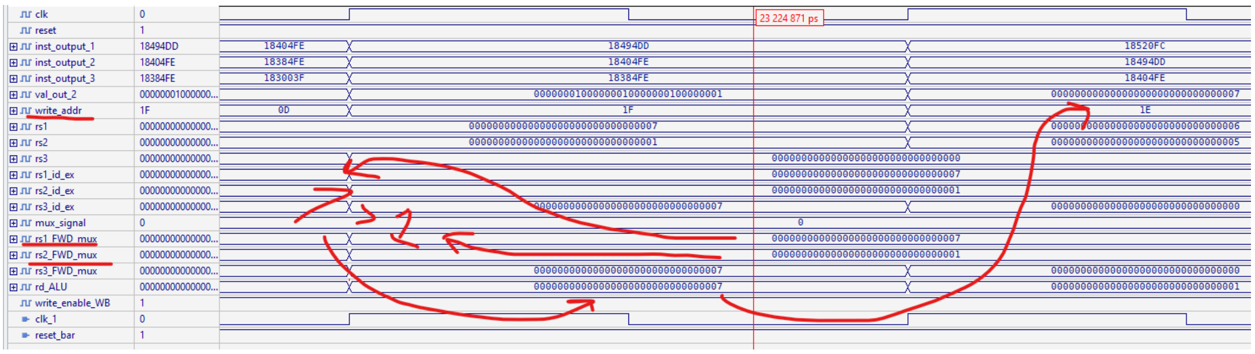
With BCW, (inst = 183003F),

(Instruction rs2, rs1, rd)

```
BCW $0, $1, $31
```

The least significant word of $1 (which is 1 represented in 32-bits) is copied to all of the 4 32-bit words of $31, which is clearly shown correctly in the waveform.

MAXWS



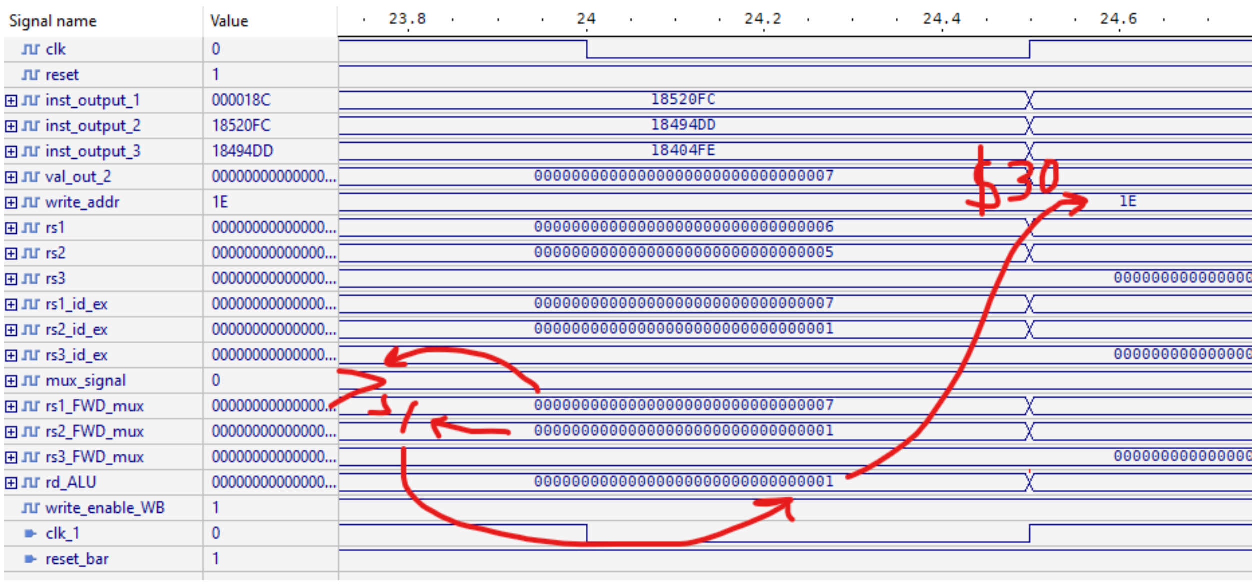
With MAXWS, (inst = 18384FE),

(Instruction rs2, rs1, rd)

```
MAXWS $1, $7, $30
```

Out of the values from the 1st word of $7 (value of 7) and $1 (value of 1), 7 > 1 so $30 correctly picks out the **highest** value in each corresponding word,in this case we only use 1 word, and since 7 is greater than 1, $30 stores 7 in its first word.

MINWS



Signal name	Value
clk	0
reset	1
inst_output_1	000018C
inst_output_2	18520FC
inst_output_3	18494DD
val_out_2	00000000000000...
write_addr	1E
rs1	00000000000000...
rs2	00000000000000...
rs3	00000000000000...
rs1_id_ex	00000000000000...
rs2_id_ex	00000000000000...
rs3_id_ex	00000000000000...
mux_signal	0
rs1_FWD_mux	00000000000000...
rs2_FWD_mux	00000000000000...
rs3_FWD_mux	00000000000000...
rd_ALU	00000000000000...
write_enable_WB	1
clk_1	0
reset_bar	1

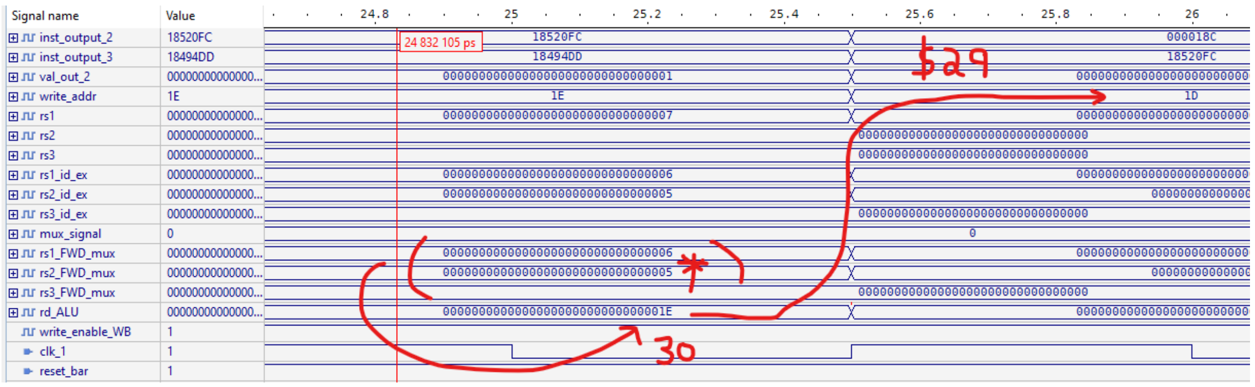
With MINWS, (inst = 18404FE),

(Instruction rs2, rs1, rd)

```
MINWS $1, $7, $30
```

Out of the values from the 1st word of $7 (value of 7) and $1 (value of 1), 7 > 1 so $30 correctly picks out the **lowest** value in each corresponding word,in this case we only use 1 word, and since 1 is lower than 7, $30 stores 1 in its first word.

MLHU



Signal name	Value
inst_output_2	18520FC
inst_output_3	18494DD
val_out_2	00000000000000...
write_addr	1E
rs1	00000000000000...
rs2	00000000000000...
rs3	00000000000000...
rs1_id_ex	00000000000000...
rs2_id_ex	00000000000000...
rs3_id_ex	00000000000000...
mux_signal	0
rs1_FWD_mux	00000000000000...
rs2_FWD_mux	00000000000000...
rs3_FWD_mux	00000000000000...
rd_ALU	00000000000000...
write_enable_WB	1
clk_1	1
reset_bar	1

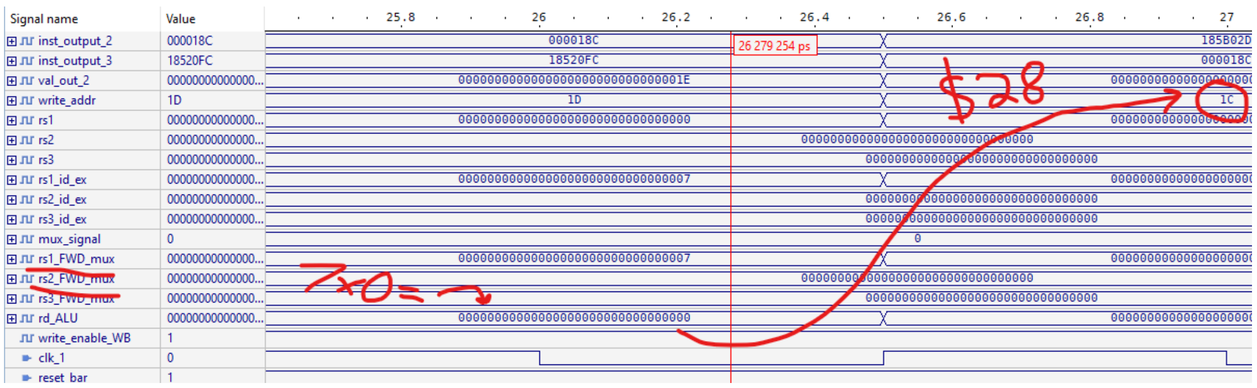
With MLHU, (inst = 18494DD),

(Instruction rs2, rs1, rd)

```
MLHU $5, $6, $29
```

The instruction multiplies the 16 rightmost bits of each of the 4 32-bit corresponding words of $5 and $6 together (with $6 having the decimal values of 0, 0, 0, 6 (from most to least significant words) and $5 having the decimal values of 0, 0, 0, 5). Their products are correctly computed and placed in the corresponding word slots in $29, with the decimal values of 0, 0, 0, and 30 (from most to least significant).

MLHSS



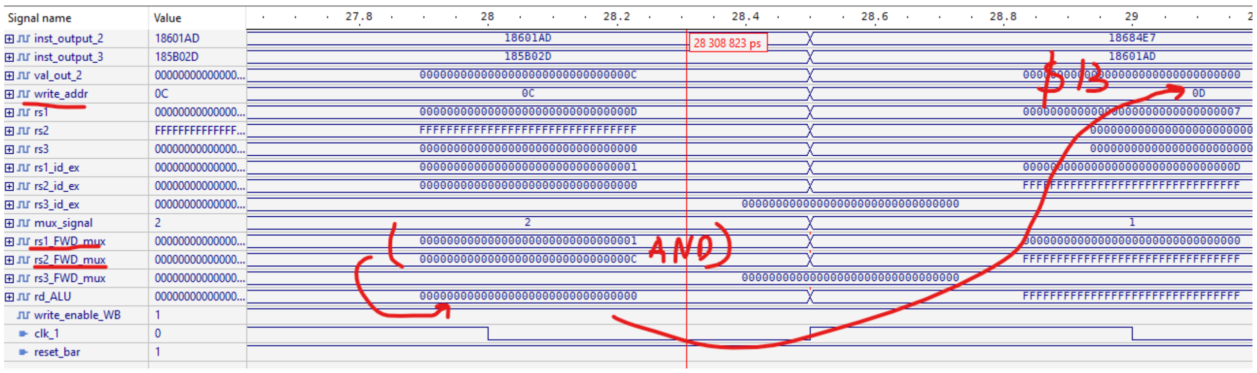
Signal name	Value
inst_output_2	000018C
inst_output_3	18520FC
val_out_2	0000000000000...
write_addr	1D
rs1	0000000000000...
rs2	0000000000000...
rs3	0000000000000...
rs1_id_ex	0000000000000...
rs2_id_ex	0000000000000...
rs3_id_ex	0000000000000...
mux_signal	0
rs1_FWD_mux	0000000000000...
rs2_FWD_mux	0000000000000...
rs3_FWD_mux	0000000000000...
rd_ALU	0000000000000...
write_enable_WB	1
clk_1	0
reset_bar	1

With MLHSS, (inst = 18520FC),
(Instruction rs2, rs1, rd)

```
MLHSS $8, $7, $28
```

We are multiplying each halfword of $7 by the sign of the corresponding halfword in $8 (in essence taking the 2’s complement of the number in $7, and adjusting it for saturated rounding, and if a halfword in $8 has the value of 0 (this value returns a result of zero in the corresponding halfword in $28)). In our case, $8 is filled with 0 so we have 0 * 7 = 0 in $28..

AND



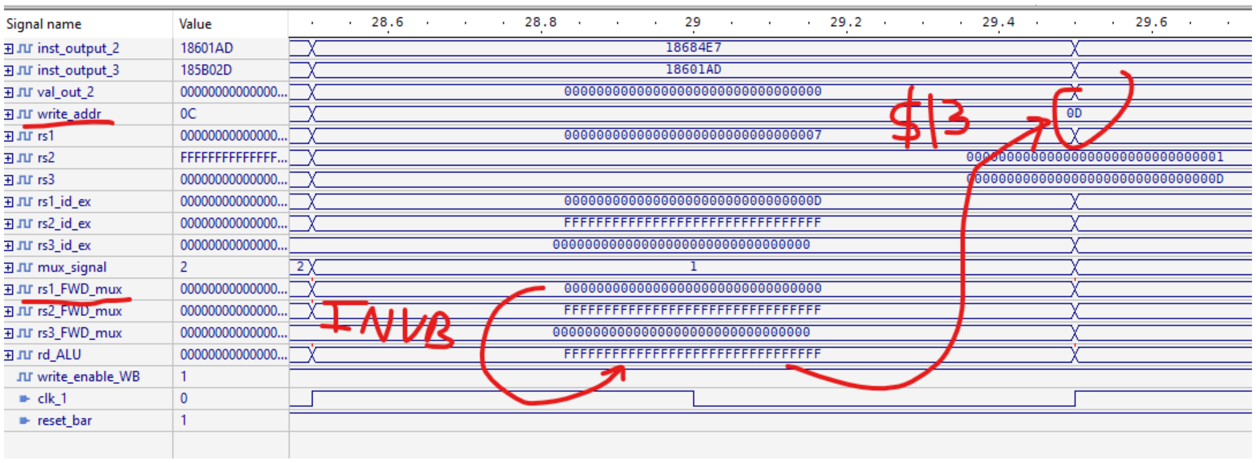
Signal name	Value
inst_output_2	18601AD
inst_output_3	185B02D
val_out_2	0000000000000...
write_addr	0C
rs1	0000000000000...
rs2	FFFFFFFFFFFFF...
rs3	0000000000000...
rs1_id_ex	0000000000000...
rs2_id_ex	0000000000000...
rs3_id_ex	0000000000000...
mux_signal	2
rs1_FWD_mux	0000000000000...
rs2_FWD_mux	0000000000000...
rs3_FWD_mux	0000000000000...
rd_ALU	0000000000000...
write_enable_WB	1
clk_1	0
reset_bar	1

With AND, (inst = 185B02D),
(Instruction rs2, rs1, rd)

```
AND $12, $1, $13
```

The logical AND of $1 with a value of 1 (0001) amd $12 with a value of 12 (1100) is 0 (0000) and this gets stored in $13

INVB



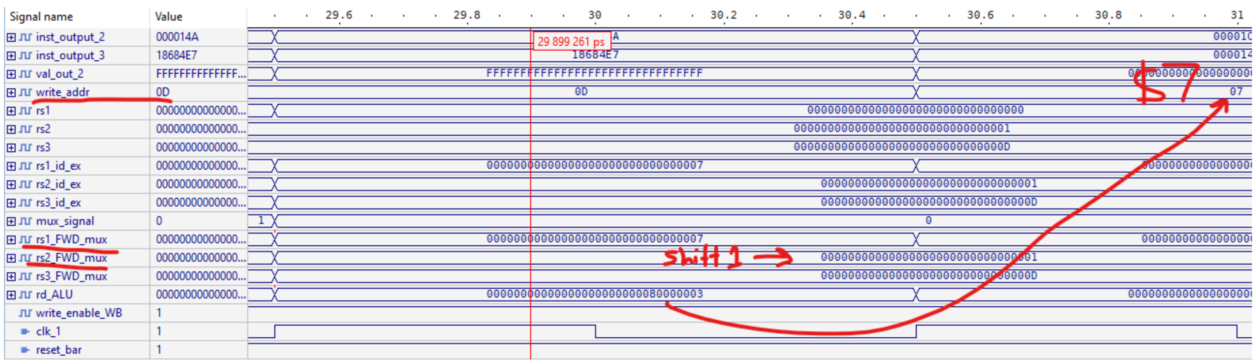
With INVB, (inst = 18601AD),

(Instruction rs2, rs1, rd)

```
INVB $0, $13, $13
```

All of the bits of $13 are flipped, where if $13 has all 1’s in the register, they will all have to become 0’s as the command’s output, in this case we go from all zeros to ones and store it back into $13.

ROTW



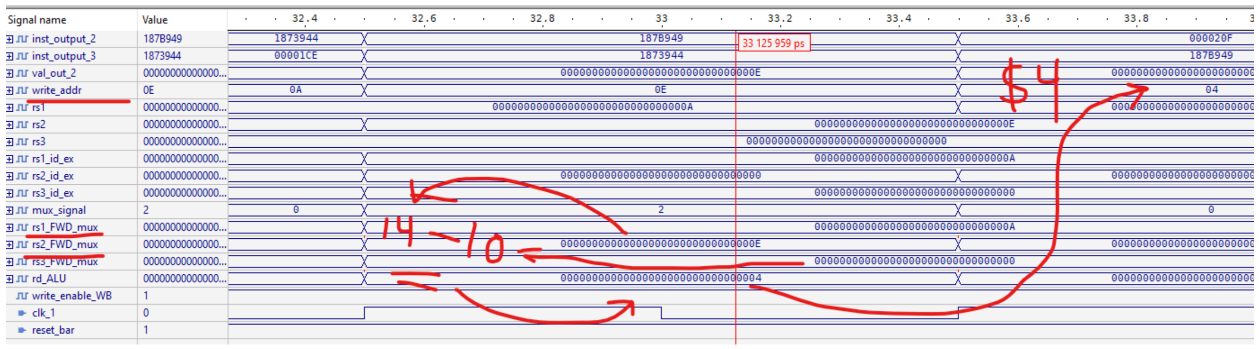
With ROTW, (inst = 18684E7),

(Instruction rs2, rs1, rd)

```
ROTW $1, $7, $7
```

$7 has a value of 7 and we are rotating by a value of one stored in $1. After rotating and we have 1000 0000 0000 0000 0000 0000 0000 0011 stored back into $7.

SFWU



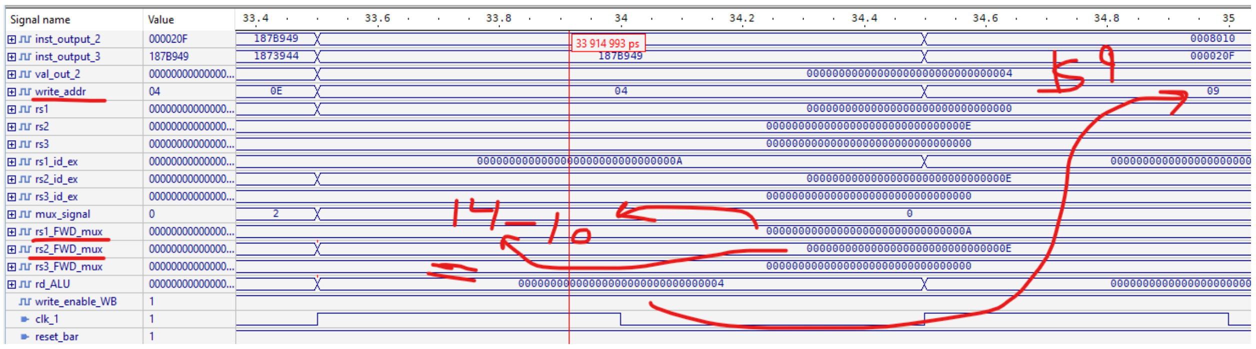
With SFWU, (inst = 1873944),

(Instruction rs2, rs1, rd)

```
SFWU $14, $10, $4
```

The difference of the corresponding words of $14 from $10 are outputted to $4, where in our waveform, we subtract the values 14 - 10 to get an output of 4 in $4.

SFHS



With SFHS (inst = 187B949),

(Instruction rs2, rs1, rd)

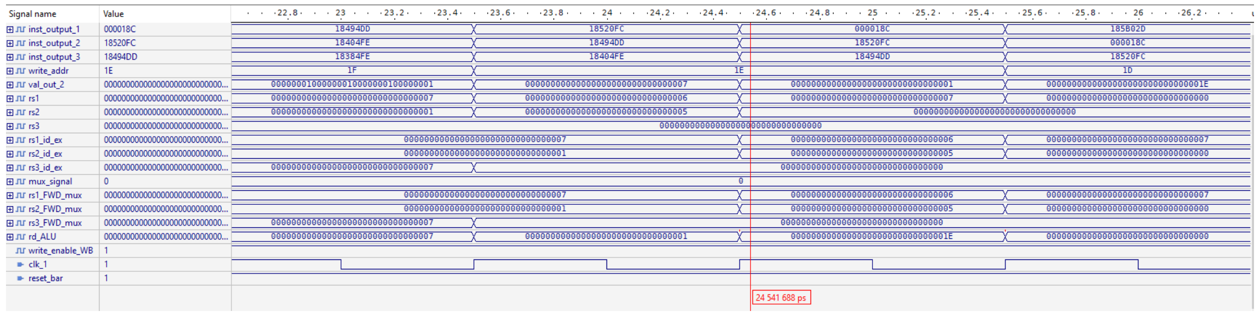
```
SFHS $14, $10, $9
```

We do the packed 16-bit halfword subtraction between $14 and $10, and use saturation rounding if needed for the difference. Since we only do 14 - 10 saturation is not necessary and the result of 4 is placed into $9.

Simulation Results

Waveform:

Below is a snippet of the waveform from our simulation:



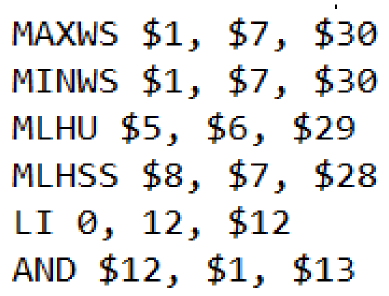
Signal name	Value
inst_output_1	000018C
inst_output_2	18520FC
inst_output_3	18494DD
write_addr	1E
val_out_2	000000000000000000000000000...
rs1	000000000000000000000000000...
rs2	000000000000000000000000000...
rs3	000000000000000000000000000...
rs1_id_ex	000000000000000000000000000...
rs2_id_ex	000000000000000000000000000...
rs3_id_ex	000000000000000000000000000...
mux_signal	0
rs1_FWD_mux	000000000000000000000000000...
rs2_FWD_mux	000000000000000000000000000...
rs3_FWD_mux	000000000000000000000000000...
rd_ALU	000000000000000000000000000...
write_enable_WB	1
clk_1	1
reset_bar	1

As explained from before, the signals “inst_output_1”, “inst_output_2”, and “inst_output_3” represent the instruction being processed in the IF, ID, and EX stages respectively. “write_addr” is the address of the register that will be written back to the Register File, which would happen during the WB stage. “val_out_2” represents the 128-bit value that will update the contents of the register specified by the “write_addr”, which is the same as the output of the EX/WB register. The signals “rs1”, “rs2”, and “rs3” represent the 128-bit outputs from the Register File, representing the contents specified by rs1, rs2, and rs3 from the instruction in “inst_output_2”. “rs1_id_ex”, “rs2_id_ex”, and “rs3_id_ex” represent the input 128-bit values of rs1, rs2, and rs3 that go to the Forwarding Muxes. The values in these signals are specified by the instruction in the “inst_output_3” signal.

“mux_signal” represents the mux signal produced by the Data Forwarding Unit that decides which of the 3 inputs (rs1, rs2, and rs3) that need the most updated value with data forwarding. “rs1_FWD_mux”, “rs2_FWD_mux”, and “rs3_FWD_mux” represent the values for rs1, rs2, and rs3 after data forwarding occurs. These signals are also the outputs from the Forwarding Muxes and the inputs into the ALU. “rd_out” represents the 128-bit result from the ALU. “write_enable_WB” represents the write enable signal that signifies if a value from the ALU should be written back to the Register File. This signal is almost always set to ‘1’, except for NOP instructions, since NOP instructions do nothing in the processor, and as such do not need to be written back to the Register File. Finally, “clk_1” and “reset_bar” represent the clock and reset signals to the processor.

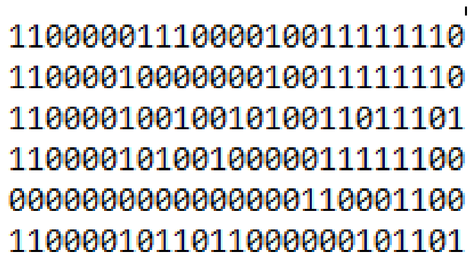
MIPS Instructions and Binary Machine Code:

Below is a snippet of the MIPS Instructions that correlate with the instructions from the waveform. From the waveform, the instruction in “inst_output_3” that has the Hexadecimal value of “18384FE” is the instruction “MAXWS $1, $7, $30”, which is the “Max Signed Word” instruction. Its binary machine code is also below, corresponding to the first row of 25 binary values in the snippet below. The next instruction in the MIPS instructions below corresponds to the “inst_output_2” with the Hex value of “18404FE” and so on.



```
MAXWS $1, $7, $30
MINWS $1, $7, $30
MLHU $5, $6, $29
MLHSS $8, $7, $28
LI 0, 12, $12
AND $12, $1, $13
```

MIPS Instructions

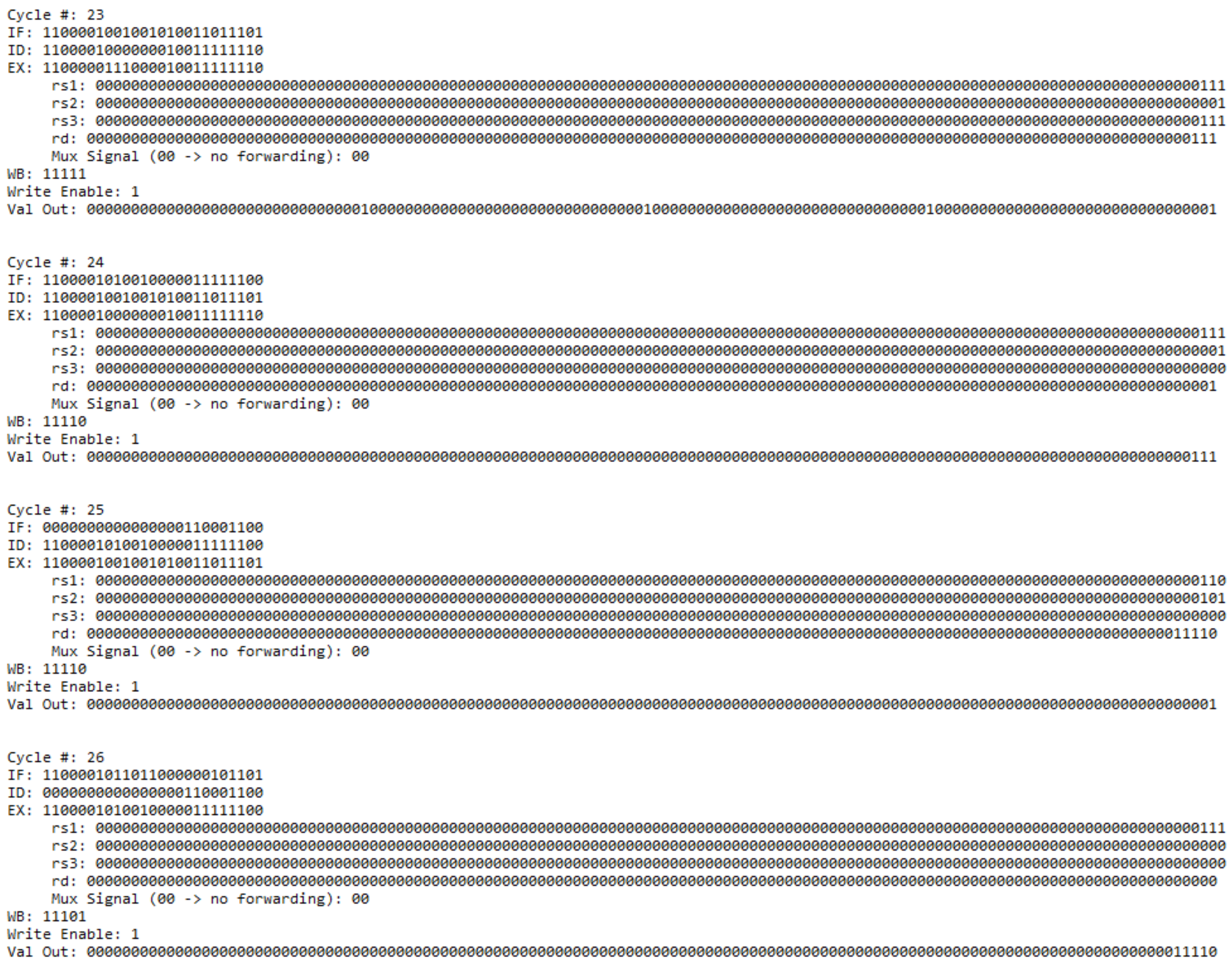


```
1100000111000010011111110
1100001000000010011111110
1100001001001010011011101
1100001010010000011111100
0000000000000000110001100
1100001011011000000101101
```

Binary Machine Code

Results File:

Below is a snippet of the Results File (results.txt) where the four cycles from the waveform can be seen below:



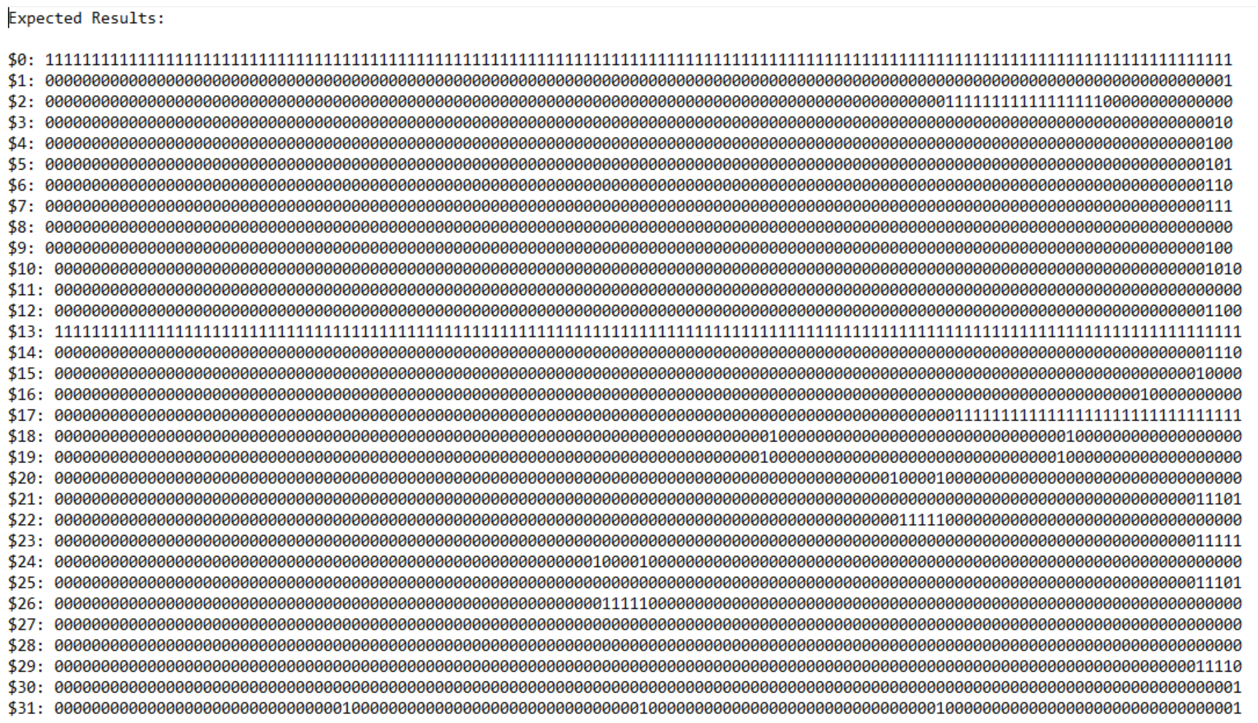
```
Cycle #: 23
IF: 1100001001001010011011101
ID: 1100001000000010011111110
EX: 1100000111000010011111110
     rs1: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000111
     rs2: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001
     rs3: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000111
     rd: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000111
     Mux Signal (00 -> no forwarding): 00
WB: 11111
Write Enable: 1
Val Out: 00000000000000000000000000000001000000000000000000000000000000010000000000000000000000000000000100000000000000000000000000000001

Cycle #: 24
IF: 1100001010010000011111100
ID: 1100001001001010011011101
EX: 1100001000000010011111110
     rs1: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000111
     rs2: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001
     rs3: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
     rd: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001
     Mux Signal (00 -> no forwarding): 00
WB: 11110
Write Enable: 1
Val Out: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000111

Cycle #: 25
IF: 0000000000000000110001100
ID: 1100001010010000011111100
EX: 1100001001001010011011101
     rs1: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000110
     rs2: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000101
     rs3: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
     rd: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011110
     Mux Signal (00 -> no forwarding): 00
WB: 11110
Write Enable: 1
Val Out: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001

Cycle #: 26
IF: 1100001011011000000101101
ID: 0000000000000000110001100
EX: 1100001010010000011111100
     rs1: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000111
     rs2: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
     rs3: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
     rd: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
     Mux Signal (00 -> no forwarding): 00
WB: 11101
Write Enable: 1
Val Out: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011110
```

As mentioned from before, the “IF”, “ID”, “EX”, and “WB” fields represent the instructions present in the IF, ID, and EX stages, along with the address of the register that will be written back to the Register File in the WB stage. “rs1”, “rs2”, and “rs3” fields represent the contents of the registers specified by rs1, rs2 and rs3 from the instruction in the EX stage. “Rd” represents the output from the ALU. “Mux Signal” represents the mux signal decided by the Data Forwarding Unit to the Forwarding Muxes, “Write Enable” represents the “write_enable” signal from the output of the EX/WB Register, and “Val Out” is the value from the “rd_out” signal from the EX/WB Register.

Going into detail about what we see in the Results File, since all of the instructions in these four cases do not cause any data hazards between themselves, the mux signal is always “00”, signifying that data forwarding was not needed for any of the instructions to be executed. The “write_enable” signal is also always set to ‘1’ since the instructions in the four cycles did not include “NOP” as an instruction. Only the “NOP” instruction is the instruction where the “write_enable” signal is set to ‘0’. Finally, in each of the cycles, the result from the ALU (or the result of the instruction in the EX stage) is represented by the “rd” field, and the result from the EX/WB Register (or the result of the instruction in the WB stage) is represented by the “Val Out” field. **(Since we set the reset signal to be ‘0’ for two cycles, the first cycle of execution from the processor starts at Cycle #3 in the Results File.)**

Expected Results File:

Below is our Expected Results (expected_results.txt) for the values in the 32 registers in the Register File once all of the 64 instructions have run:



```
Expected Results:

$0: 11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
$1: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001
$2: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001111111111111111100000000000000
$3: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010
$4: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000100
$5: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000101
$6: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000110
$7: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000111
$8: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
$9: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000100
$10: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001010
$11: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
$12: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001100
$13: 11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
$14: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001110
$15: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010000
$16: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010000000000
$17: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001111111111111111111111111111111
$18: 00000000000000000000000000000000000000000000000000000000000000000000000000000100000000000000000000000000000001000000000000000000
$19: 00000000000000000000000000000000000000000000000000000000000000000000000000001000000000000000000000000000000010000000000000000000
$20: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010000100000000000000000000000000000000
$21: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011101
$22: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001111100000000000000000000000000000000
$23: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011111
$24: 00000000000000000000000000000000000000000000000000000000001000010000000000000000000000000000000000000000000000000000000000000000
$25: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011101
$26: 00000000000000000000000000000000000000000000000000000000000111110000000000000000000000000000000000000000000000000000000000000000
$27: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
$28: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
$29: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000011110
$30: 00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001
$31: 00000000000000000000000000000001000000000000000000000000000000010000000000000000000000000000000100000000000000000000000000000001
```

Comparing the results from the Results File to the expected results, we see that from the waveform, only the first 3 MIPS instructions were executed in the EX stage. As such, we can compare the values of the “rd” registers for these 3 instructions with our expected results.

Looking at the first two instructions, since they both use $30 as their “rd” register, we can only compare the value of $30 after the “MINWS $1, $7, $30” instruction is executed. After this instruction was executed, the value of $30 from the ALU was “000000000000000000000000000000000000000000000000000000000000000000000000000 00000000000000000000000000000000000000000000000000001” from the rd register in Cycle #24. This is the exact same as the value of register $30 from our expected results file. With the 3rd MIPS instruction (“MLHU $5, $6, $29”) that we can see that executed from the waveform, the value of $29 from the ALU was “000000000000000000000000000000000000000000000000000000000000000000000000000 00000000000000000000000000000000000000000000000011110”, which is the same as the our expected value for the value of $29 at the end of execution.

Conclusion

Working on this project gave us a lot of insight on how pipelining inside a processor actually works. Data forwarding as a concept seems very challenging to understand and gave us a hard time thinking about how to implement it. When we finally realized how to create the unit, it was a lot easier than we initially thought. Despite some challenges, our Processor achieves all the goals that were set according to the results we printed and read. If we had more time we could improve our data efficiency as passing the whole 25-bit instruction through the processor is very inefficient.