Pipelined SIMD Multimedia Unit Design

Stony Brook University

Farhaan Khan & Johnson Varghese

Mikhail Dorojevets

December 1, 2024

Pipelined SIMD Multimedia Unit Design

The purpose of this project is to design a pipelined SIMD multimedia unit using VHDL/Verilog hardware description languages. The project aims to explore the structural and behavioral design of a four-stage pipelined multimedia unit capable of executing a reduced set of multimedia instructions, similar to those found in the Sony Cell SPU and Intel SSE architectures. This project involves designing a four-stage pipelined multimedia ALU unit that supports various multimedia operations. The unit will be tested using a set of multimedia instructions that operate on subword parallelism, which is commonly used in SIMD (Single Instruction, Multiple Data) architectures. The design will include data forwarding mechanisms to resolve pipeline hazards and ensure efficient execution of instructions.

The project is divided into two parts:

- Part 1: Develop and verify the VHDL source code for all multimedia ALU functions at the third stage.
- Part 2: Complete the full design of the pipelined multimedia unit, including testbenches, block diagrams, and simulation results for all stages of the pipeline.

Part 1: Multimedia ALU

Design Procedure

To implement the ALU we first established which inputs and outputs the ALU could take. From the specifications of the project, we know that the ALU should take up to three inputs from the register file and generate one output. We also know that there needs to be a way for the ALU to figure out which operation to perform, so we also passed in the instruction as an input. We chose to pass in the entire instruction so that implementing certain instructions such as *load*

immediate would be easier. Thus, our entity declaration in VHDL can be shown in Figure 1 below.

Figure 1

ALU Entity Declaration in VHDL

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

entity multimedia_alu is
    port(
    instruction: in bit_vector(24 downto 0);
    rs1: in bit_vector(127 downto 0);
    rs2: in bit_vector(127 downto 0);
    rs3: in bit_vector(127 downto 0);
    rd : out bit_vector(127 downto 0)
    );
end multimedia_alu;
```

We used a behavioral style approach to implementing the architecture, and so we used a process that executes when any of our inputs is changed. The process first looks at the first bit of the instruction. If it is a 0, then we know the operation is *load immediate*. If it is a 1, then we look at the second bit. If the second bit is a 0, then we know the instruction is of R4 type. If the second bit is a 1, then we know the instruction format is of R3 type. The general format of the architecture body can be seen in Figure 2 below.

To implement each specific operation, we created a procedure for each operation. Each procedure takes in the necessary information to run the operation, and then writes the result to the output register. The implementation of the *load immediate* procedure can be shown below in Figure 3. All of the R4 instructions were implemented in one procedure, shown in Figure 3. The "Long/Int", "Subtract/add", and "High/low" bits were passed as inputs to the procedure, and so with some arithmetic, all 8 of those instructions were able to be implemented in one procedure.

The rest of the R3 instructions were all implemented in their own procedures similarly to load immediate. For instructions that worked with signed and unsigned numbers, we would convert from the bit_vector type to std_logic_vector, and then from std_logic_vector to signed or unsigned.

To implement saturation for any of the instructions, we would define constants that would give us the minimum and maximum values for data of whatever size we were working with (16/32/64 bits). Then, we would store the final result of our calculations in a vector that was 1 bit larger than the size we were working with. We would then compare the value of that final result with our constants, and assign the proper value to our output register based on the result of the comparison.

Figure 2

Architecture Body Structure

```
--------------------------- Architecture Body -----------------------------------

begin
    process(rs1, rs2, rs3, instruction)

    variable rd_p : bit_vector(127 downto 0);

    begin
        -- Figure out instruction type

        -- Load
        if (instruction(24) = '0') then
            load_immediate(rs1, instruction(23 downto 21), instruction(20 downto 5), rd_p);
            rd <= rd_p;

        -- R4 Instruction Type (Multiply-add/subtract)
        elsif (instruction(23) = '0') then
            r4_instruction(signed(to_stdlogicvector(rs1)), signed(to_stdlogicvector(rs2)),
            signed(to_stdlogicvector(rs3)), unsigned(to_stdlogicvector(instruction(22 downto 20))), rd_p);
            rd <= rd_p;

        -- R3 Instruction Type
        else

            --SLHI
            if (instruction(18 downto 15) = "0001") then
                shift_left_halfword_immediate(rs1, instruction(13 downto 10), rd_p);
                rd <= rd_p;

            -- AU
            elsif (instruction(18 downto 15) = "0010") then
                add_word_unsigned(unsigned(to_stdlogicvector(rs1)), unsigned(to_stdlogicvector(rs2)), rd_p);
                rd <= rd_p;

            --CNT1H
```

Figure 3

Implementing Load Immediate Operation

```
-----------------Load Immediate---------------------------------------
procedure load_immediate (rs1, load_index, immediate: in bit_vector; rd: out bit_vector) is
     -- rs1 has rd, rs2 has 16-bit immediate, rs3 has load index
     variable index : natural;

begin

     -- Get index
     index := bits_to_natural(load_index);

     --Read rd
     rd := rs1;

     -- Load immediate
     rd( ((index * 16) + 15) downto (index * 16)) := immediate;

end load_immediate;
```

Figure 4

Implementing R4 Instructions

```
-------------------R4 Instruction Type- multiply and add/subtract---------------------------------------
procedure r4_instruction (rs1, rs2, rs3: in signed(127 downto 0);
                          specs: in unsigned(2 downto 0);
                          rd: out bit_vector(127 downto 0)) is

variable size : integer := 32 + (32 * to_integer(unsigned(specs(2 downto 2))));
variable factor1 : signed(size/2 - 1 downto 0);
variable factor2: signed(size/2 - 1 downto 0);
variable term: signed(size - 1 downto 0);
variable product: signed(size - 1 downto 0);
variable result: signed(size downto 0);
variable current_lower_index : integer;

constant MAX_32 : signed(31 downto 0) := X"7FFFFFFF";
constant MIN_32 : signed(31 downto 0) := X"80000000";
constant MAX_64 : signed(63 downto 0) := X"7FFFFFFFFFFFFFFF";
constant MIN_64 : signed(63 downto 0) := X"8000000000000000";

begin

    -- For each pack of size 'size' (32 or 64) in 127-bit registers loop
    for i in 0 to 3 - 2 * to_integer(unsigned(specs(2 downto 2))) loop

        -- Calculate lower index
        current_lower_index := (i * size + (size/2 * to_integer(unsigned(specs(0 downto 0)))));

        -- Get current factors
        factor1 := rs3(current_lower_index + (size/2) - 1 downto current_lower_index);
        factor2 := rs2(current_lower_index + (size/2) - 1 downto current_lower_index);

        -- Compute multiplication
        product := factor1 * factor2;

        -- Get addition/subtraction term
        term := rs1((i * size) + size - 1 downto i * size);

        -- If we are subtracting --
        if (specs(1) = '1') then
            result := resize(term, size + 1) - resize(product, size + 1);

        -- If we are adding --
        else
            result := resize(term, size + 1) + resize(product, size + 1);
        end if;

        -- Saturation For Long --
        if (size = 64) then
            if (result > MAX_64) then
                rd((i * size) + size - 1 downto i * size) := to_bitvector(std_logic_vector(MAX_64));
            elsif (result < MIN_64) then
                rd((i * size) + size - 1 downto i * size) := to_bitvector(std_logic_vector(MIN_64));
            else
                rd((i * size) + size - 1 downto i * size) := to_bitvector(std_logic_vector(resize(result, size)));
            end if;

        -- Saturation For Int --
        else
            if (result > MAX_32) then
                rd((i * size) + size - 1 downto i * size) := to_bitvector(std_logic_vector(MAX_32));
            elsif (result < MIN_32) then
                rd((i * size) + size - 1 downto i * size) := to_bitvector(std_logic_vector(MIN_32));
            else
                rd((i * size) + size - 1 downto i * size) := to_bitvector(std_logic_vector(resize(result, size)));
            end if;
        end if;

    end loop;
end r4_instruction;
```

Figure 5

Implementing R3 Instructions

```
-----------------------------------R3 Instructions ------------------------------------------------

------------------- SLHI
procedure shift_left_halfword_immediate (rs1: in bit_vector(127 downto 0); shift: in bit_vector(3 downto 0);
                                         rd: out bit_vector(127 downto 0))  is

variable shift_amt : natural := bits_to_natural(shift);
variable lower_index : integer;
variable halfword : bit_vector(15 downto 0);

begin

    -- for each halfword
    for i in 0 to 7 loop
        lower_index := i * 16;
        -- read halfword in rs1
        halfword := rs1(lower_index + 15 downto lower_index);
        -- shift amount
        for j in 0 to shift_amt - 1 loop
            -- shift each value in halfword
            for k in 15 downto 1 loop
                halfword(k) := halfword(k - 1);
            end loop;
            -- set LSB to 0
            halfword(0) := '0';
        end loop;

        -- place result in destination register
        rd(lower_index + 15 downto lower_index) := halfword;
    end loop;

end shift_left_halfword_immediate;


------------------- AU
procedure add_word_unsigned (rs1, rs2: in unsigned(127 downto 0); rd: out bit_vector(127 downto 0)) is

variable sum: unsigned(31 downto 0);
variable lower_index: integer;

begin
```

Testing Our Results

To test our instructions, we applied values to our input registers and to the instruction register. For simplicity, we generally applied and displayed hexadecimal values. We would then wait one clock period (10 ns), and check the waveform of our output register. Since the operations would work on packs of 16, 32, or 64 bits, we treated each pack of bits as a test case and applied different values to them. For the figures that will follow, the top row of our simulation waveform depicts the instruction register. The next four rows display the contents of

rs1, rs2, rs3, and rd, respectively. For R3 instruction types, rs3 (and sometimes rs2) can be ignored.

Load Immediate

The output of our *load immediate* program (shown in Figure 6) takes the opcode 00000000000000000001100000 where rs1 is initialized to 0. From the opcode, we see that bit 24 is 0 and the load index (bits 21-23) are also 0. Also, our immediate value (bits 20-5) gives us a value of 3. Based on this information, we expect 3 to be loaded into index 0 of rd which matches our simulation results shown in Figure 6.

Figure 6

Testing Load Immediate

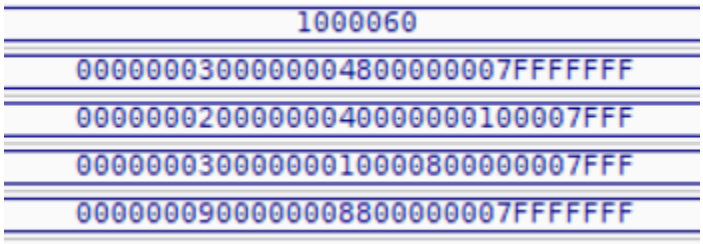
Signed Integer Multiply-Add Low With Saturation

The output of our *signed integer multiply add-low with saturation* (shown below) uses the opcode 10000 where 10 implies an R4 instruction type and 000 executes a signed integer multiply add-low with saturation procedure. The first execution of this function give us (7FFF*7FFF)+7FFFFFFF which results in a positive overflow. This matches the first 8 hex values in our output waveform. The second execution is (8000*0001)+ 80000000 which is a very negative number plus another negative number which results in negative overflow. This matches the next 8 hex values in our output. For the 3rd word we did (0004*0001)+00000004 which is 4+4 which equals 8 and this also matches the third word in our output register. Lastly, the fourth

word does (0003*0002)+00000003 which is 6+3 which equals 9 and this also matches the output value in the fourth word.

Figure 7

Testing Signed Integer Multiply-Add Low With Saturation

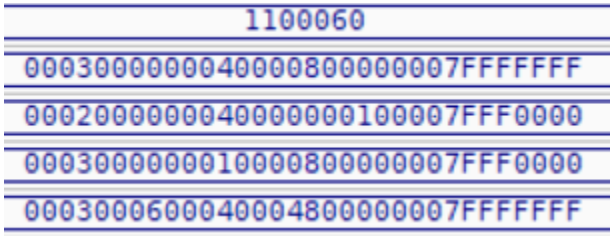


Signed Integer Multiply-Add High With Saturation

The output of our *signed integer multiply add-high with saturation* (shown below) uses the opcode 10001 where 10 implies an R4 instruction type and 001 executes a signed integer multiply add-high with saturation procedure. The first execution of this function give us (7FFF*7FFF)+7FFFFFFF which results in a positive overflow. This matches the first 8 hex values in our output waveform. The second execution is (8000*0001)+ 80000000 which is a very negative number plus another negative number which results in negative overflow. This matches the next 8 hex values in our output. For the 3rd word we did (0004*0001)+00040000 which is 00040004 and this matches the third word in our output register. Lastly, the fourth word does (0003*0002)+00030000 which is 00030006 and this also matches the output value in the fourth word.

Figure 8

Testing Signed Integer Multiply-Add High With Saturation

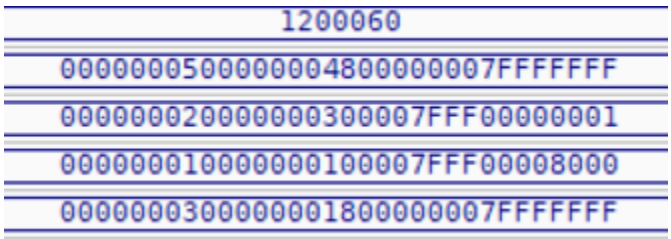


Signed Integer Multiply-Subtract Low With Saturation:

The output of our *signed integer multiply subtract-low with saturation* (shown below) uses the opcode 10010 where 10 implies an R4 instruction type and 010 executes a signed integer multiply subtract-low with saturation procedure. The first execution of this function give us 7FFFFFFF- (8000*0001) which results in a positive overflow because it's the positive maximum plus the maximum negative value. This matches the first 8 hex values in our output waveform. The second execution is 80000000 - (7FFF*7FFF) which is a very negative number minus a high positive number which results in negative overflow. This matches the next 8 hex values in our output. For the 3rd word we did 00040000 - (0003*0001) which is 00000001 and this matches the third word in our output register. Lastly, the fourth word does 00000005 - (0001*0002) which is 00000003 and this also matches the output value in the fourth word.

Figure 9

Testing Signed Integer Multiply-Subtract Low With Saturation

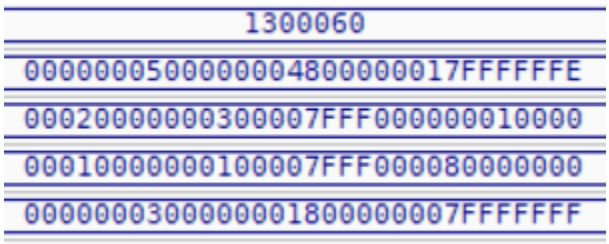


Signed Integer Multiply-Subtract High With Saturation:

The output of our *signed integer multiply subtract-high with saturation* (shown below) uses the opcode 10011 where 10 implies an R4 instruction type and 011 executes a signed integer multiply subtract-high with saturation procedure. The first execution of this function give us 7FFFFFFE- (8000*0001) which results in a positive overflow because it's the positive maximum plus the maximum negative value. This matches the first 8 hex values in our output waveform. The second execution is 80000001 - (7FFF*7FFF) which is a very negative number minus a high positive number which results in negative overflow. This matches the next 8 hex values in our output. For the 3rd word we did 00040000 - (0003*0001) which is 00000001 and this matches the third word in our output register. Lastly, the fourth word does 00000005 - (0001*0002) which is 00000003 and this also matches the output value in the fourth word.

Figure 10

Testing Signed Integer Multiply-Subtract High With Saturation



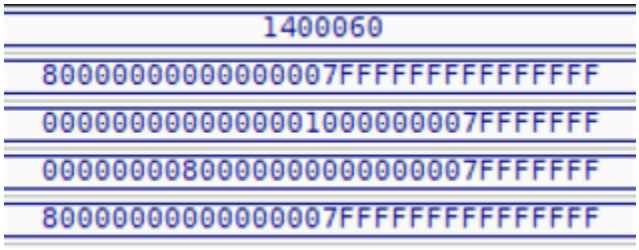
Signed Long Multiply-Add Low With Saturation

The output of our *signed long multiply add-low with saturation* (shown below) uses the opcode 10100 where 10 implies an R4 instruction type and 100 executes a signed long multiply add-low with saturation procedure. The first execution of this function give us (7FFFFFFF*7FFFFFFF)+7FFFFFFFFFFFFFFF which results in a positive overflow. This matches the first 16 hex values in our output waveform. The second execution is

(80000000*00000001)+ 8000000000000000 which is a very negative number plus another negative number which results in negative overflow.

Figure 11

Testing Signed Long Multiply-Add Low With Saturation

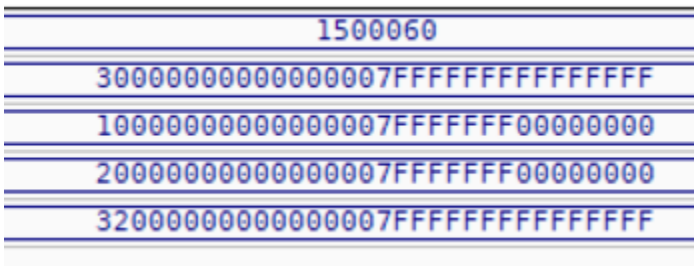


Signed Long Multiply-Add High With Saturation

The output of our *signed long multiply add-high with saturation* (shown below) uses the opcode 10101 where 10 implies an R4 instruction type and 101 executes a signed integer multiply add-low with saturation procedure. The first execution of this function give us (7FFFFFFF*7FFFFFFF)+7FFFFFFFFFFFFFFF which results in a positive overflow. This matches the first 16 hex values in our output waveform. The second execution is (10000000*20000000)+ 3000000000000000 which would give us 3 2 as the most significant bits which matches our simulation result.

Figure 12

Testing Signed Long Multiply-Add High With Saturation

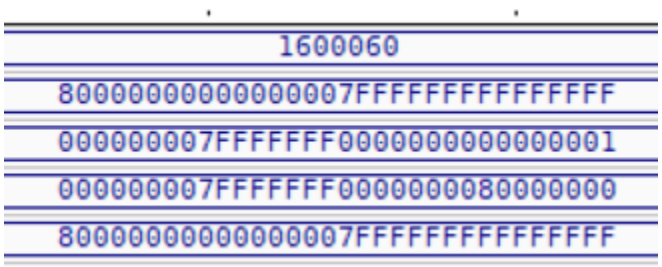


Signed Long Multiply-Subtract Low With Saturation

The output of our *signed long multiply subtract-low with saturation* (shown below) uses the opcode 10110 where 10 implies an R4 instruction type and 110 executes a signed long integer multiply subtract-low with saturation procedure. The first execution of this function give us 7FFFFFFFFFFFFFF- (80000000*00000001) which results in a positive overflow because it's the positive maximum plus the maximum negative value. This matches the first 16 hex values in our output waveform. The second execution is 8000000000000000 - (7FFFFFFF*7FFFFFFF) which is a very negative number minus a high positive number which results in negative overflow. This matches the next 16 hex values in our output.

Figure 13

Testing Signed Long Multiply-Subtract Low With Saturation

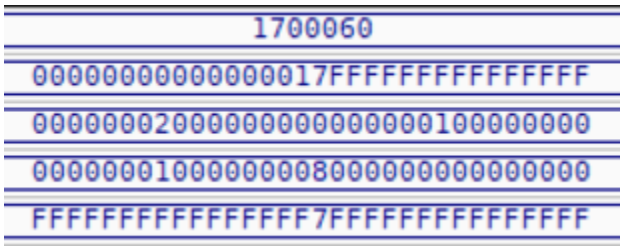


Signed Long Multiply-Subtract High With Saturation

The output of our *signed long multiply subtract-high with saturation* (shown below) uses the opcode 10111 where 10 implies an R4 instruction type and 111 executes a signed long integer multiply subtract-high with saturation procedure. The first execution of this function give us 7FFFFFFFFFFFFFF- (00000000*00000000) which results in the value in rs1. This matches the first 16 hex values in our output waveform. The second execution is 8000000000000000 - (00000000*00000000) which results in the second 16 hex values in rs1. This matches the next 16 hex values in our output.

Figure 14

Testing Signed Long Multiply-Subtract High With Saturation

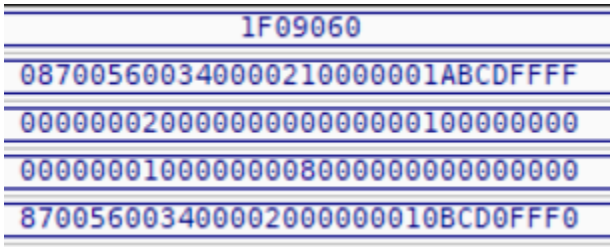


SLHI

To test *shift left halfword immediate*, we loaded a value of 4 in the four least significant bits of instruction field rs2 (bits 14 to 10). We used 4 since our results are displayed in hexadecimal, and so a value of 4 will mean that each hexadecimal digit should shift over one position to the left. We expected the corresponding halfword in rd to contain the same halfword as in rs1 shifted over by one position. Since it is a logical left shift, we expect to shift in 0 at the least significant bit. Figure 15 shows the output of our operation, which matches our expected values.

Figure 15

Testing SLHI



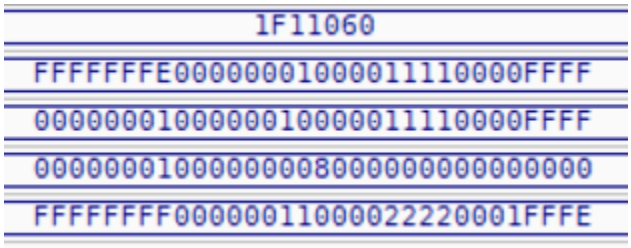
AU

To test *add word unsigned*, we chose four different pairs of values for each word. We know that FFFFFFFE + 00000001 will give FFFFFFFF, 00000001 + 00000010 will give us

0000001, and that 00001111 + 00001111 will give us 00002222, and that 0000FFFF + 0000FFFF will give us 0001FFFE. Figure 16 below shows our simulation results, all of which match our expected results.

Figure 16

Testing AU

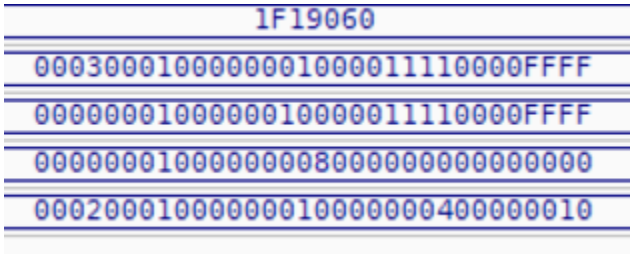


CNT1H

To test *count 1s in halfwords*, we loaded values into each halfword of rs1. For 0003, we expect a count of 2 ones. For 0001, we expect a count of 1 one. For 0000, we expect a count of 0 ones. For 1111, we expect a count of 4 ones. And finally, for FFFF we expect a count of 16 ones (10 in hexadecimal). Figure 17 below shows our simulation results, which matches our expected results.

Figure 17

Testing CNT1H

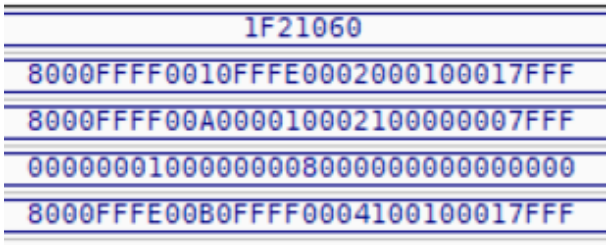


AHS

To test *add halfword saturated*, we chose eight different pairs of values for each halfword. We know that 8000 + 8000 should cause saturation which will lead to 8000. FFFF + FFFF should give us FFFE (since -1 + -1 = -2). 0010 + 00A0 should give us 00B0. FFFE + 0001 should give us FFFF. 0002 + 0002 should give us 0004. 0010 + 0000 should give us 0010. 0001 + 1000 should give us 1001. 0001 + 0000 should give us 0001. And finally, 7FFF + 7FFF should cause saturation giving us the maximum value 7FFF. Figure 18 below shows our simulation results, which matches our expected results.

Figure 18

Testing AHS

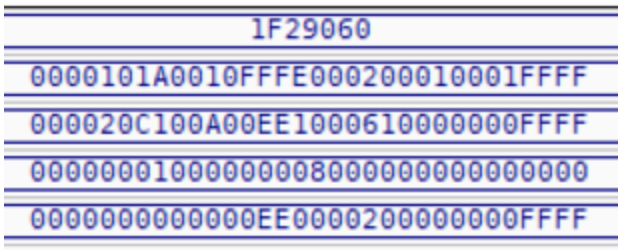


AND

To test *AND*, we did a bitwise logical AND of the contents of rs1 and rs2. Using simpler cases such as 0000 and 0000 is 0000, 0001 and 0002 is 0000, FFFF and FFFF = FFFF we can see that we get the right outputs. Figure 19 below shows our simulation results, which matches our expected results.

Figure 19

Testing AND

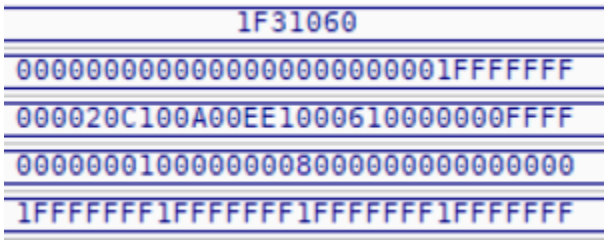


BCW

To test *broadcast word*, we loaded the rightmost word of rs1 with 1FFFFFF. Thus, we expect each word of our output register to be 1FFFFFFF. Figure 20 below shows our simulation results, which matches our expected results.

Figure 20

Testing BCW

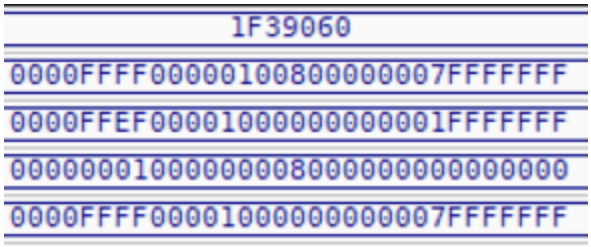


MAXWS

To test *max signed word*, we loaded four different pairs of values to the words of rs1 and rs2. Between 0000FFFF and 0000FFEF, we know 0000FFFF is greater. Between 00000100 and 00001000, we know that 00001000 is greater. Between 80000000 and 00000000, we know that 00000000 is greater (since the other value is negative). And finally, between 7FFFFFFF and 1FFFFFFF, we know 7FFFFFFF is greater. Figure 21 below shows our simulation results, which matches our expected results.

Figure 21

Testing MAXWS

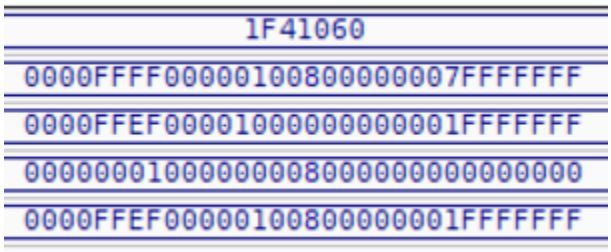


MINWS

To test *min signed word*, we used the same test cases as with *max signed word*. We know that for this operation, we should get the opposite result since we're outputting the minimum between each word of rs1 and rs2 instead of the maximum. Figure 22 below shows our simulation results, which matches our expected results.

Figure 22

Testing MINWS

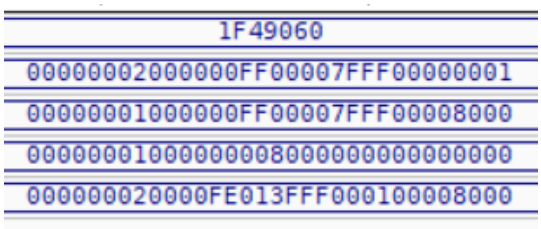


MLHU

To test *multiply low unsigned*, we loaded four different values into each word of rs1 and rs2. For 0002 * 0001, we expect the result to be 00000002. For 00FF * 00FF, we expect the result to be 0000FE01. For 7FFF * 7FFF, we expect our output to be FFF0001. For 0001 * 8000, we expect the result to be 00008000. Figure 23 below shows our simulation results, which matches our expected results.

Figure 23

Testing MLHU

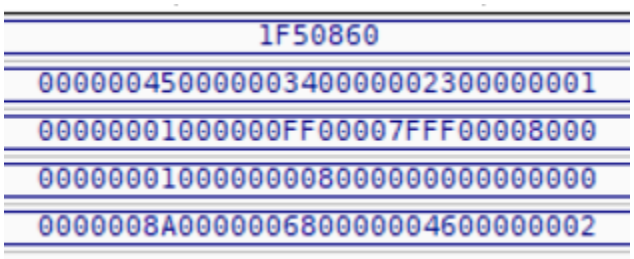


MLHCU

To test *multiply low by constant unsigned*, we loaded a value of 2 into the rs2 field of the instruction (bits 14 to 10), and different values into the lower 16 bits of each word in rs1. This makes it easy to verify our results. We expect 0045 * 2 to give us 0000008A. 0034 * 2 will give us 00000006. 0023 * 2 will give us 00000046. And finally, 0001 * 2 will give us 00000002. Figure 24 below shows our simulation results, which matches our expected results.

Figure 24

Testing MLHCU

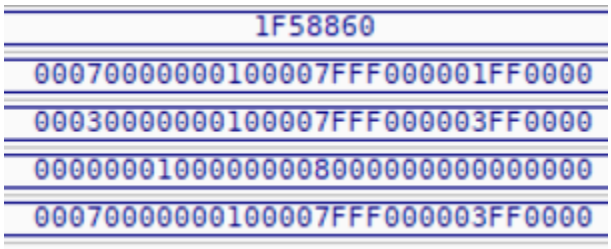


OR

To test *OR*, we did a bitwise logical OR of the contents of rs1 and rs2. Using simpler cases such as 0000 or 0000 is 0000, 01FF or 03FF is 03FF, 7FFF or 7FFF = 7FFF we can see that we get the right outputs. Figure 25 below shows our simulation results, which matches our expected results.

Figure 25

Testing OR

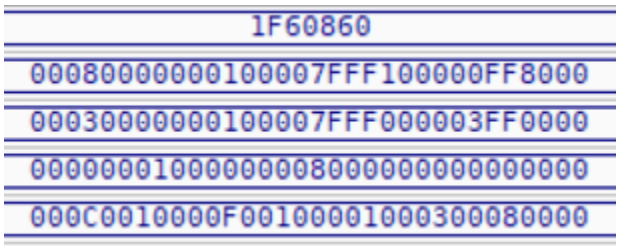


CLZH

To test *count leading zeros in halfwords*, we have to count the number of leading 0's for each halfword before a 1 and if the halfword value is 0 the value is 16. The first halfword (8000) has 0 leading 0's since the first bit is a 1 which matches our simulation. The second halfword (00FF) has 8 leading 0's because each hex value is 4 and the most significant bit of F is 1 which matches our simulation value. For the third one, the value (1000) and 1 has 3 leading 0's so the result is 3 which matches our simulation. The 4th halfword has 1 leading zero since the most significant bit of 7 is 0. The 5th halfword is our special case where we get 16 which is 0010 in hex. For the 6th halfword (0001) we get 15 because each hex value has four 0's except 1 which would give us 15 or F. The 7th halfword has the value 0 which means we default it to 16 and the final half word has 12 zeros because there are 3 0 hex values and the most significant bit of 8 is 1. Figure 26 below shows our simulation results, which matches our expected results.

Figure 26

Testing CLZH

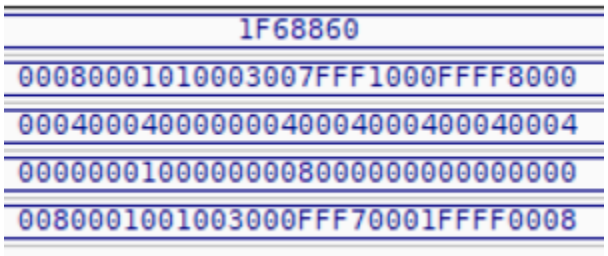


RLH

To test *rotate left bits in halfwords*, we followed a similar procedure as when we tested *shift left halfword immediate*. We loaded a value of 4 into the 4 least significant bits of each halfword in rs2. This way, we expect each halfword in our output register to be the corresponding halfword in rs1 rotated left by one hexadecimal digit. Since we are rotating left, we expect the least significant hexadecimal digit of each halfword in our output register to be equal to the most significant hexadecimal digit in the corresponding halfword of rs1 (before we rotated). Figure 27 below shows our simulation results, which matches our expected results. We know our rotation works correctly since when we rotate 7FFF to the left, we get a value of FFF7.

Figure 27

Testing RLH

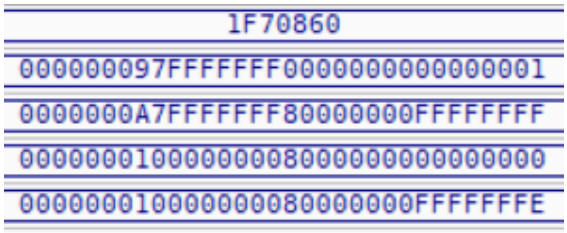


SFWU

To test *subtract word from unsigned*, we subtract the contents of rs1 and rs2 for each word in the corresponding register. In the first word we did FFFFFFFF - 00000001 which gives us FFFFFFFE which matches the value in simulation. In the second word we did 8000000 - 0000000 which gives us 8000000 and that also matches our simulated value. For the third word, we did 7FFFFFFF - 7FFFFFFF which should give 00000000 and that also matches our simulated value. Finally for the fourth word, we 0000000A - 00000009 which is 10 - 9 which gives us 00000001 and that matches our simulation as well. Figure 28 below shows our simulation results, which matches our expected results.

Figure 28

Testing SFWU

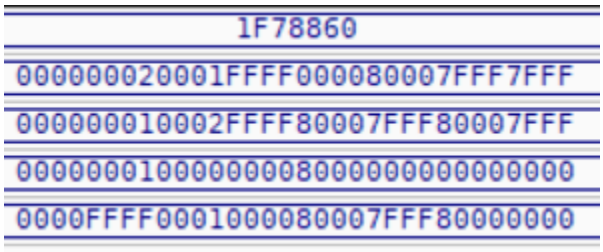


SFHS

To test *subtract from halfword saturated*, we chose eight different pairs of values for each halfword. We know that 0000 - 0000 should give us 0000. 0001 - 0002 should give us FFFF. 0002 - 0001 should give us 0001. FFFF - FFFF should give us 0000. 8000 - 0000 should give us 8000. 7FFFF - 8000 should result in saturation giving us 7FFF since we are subtracting a negative from the maximum positive value. 8000 - 7FFF should also give us saturation since we are subtracting a positive value from the lowest negative value. Finally, 7FFF - 7FFF should give us 0000. Figure 29 below shows our simulation results, which matches our expected results.

Figure 29

Testing SFHS

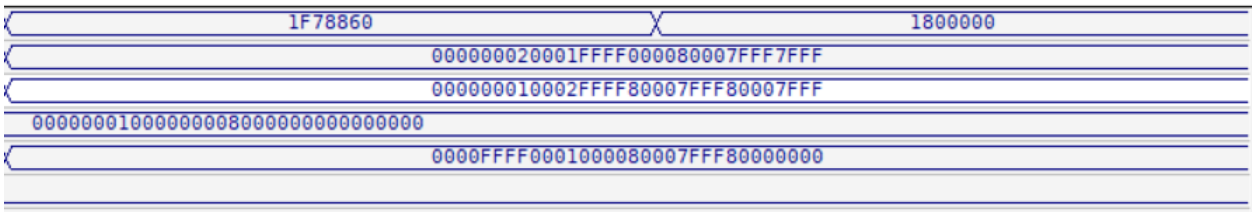


NOP

To test *NOP*, we have to make sure that nothing gets written to the register file and since our last value from Figure 29 was 0000FFFF0001000080007FFF80000000, we can see in Figure 30 that the value of rd was not modified at all.

Figure 30

Testing NOP



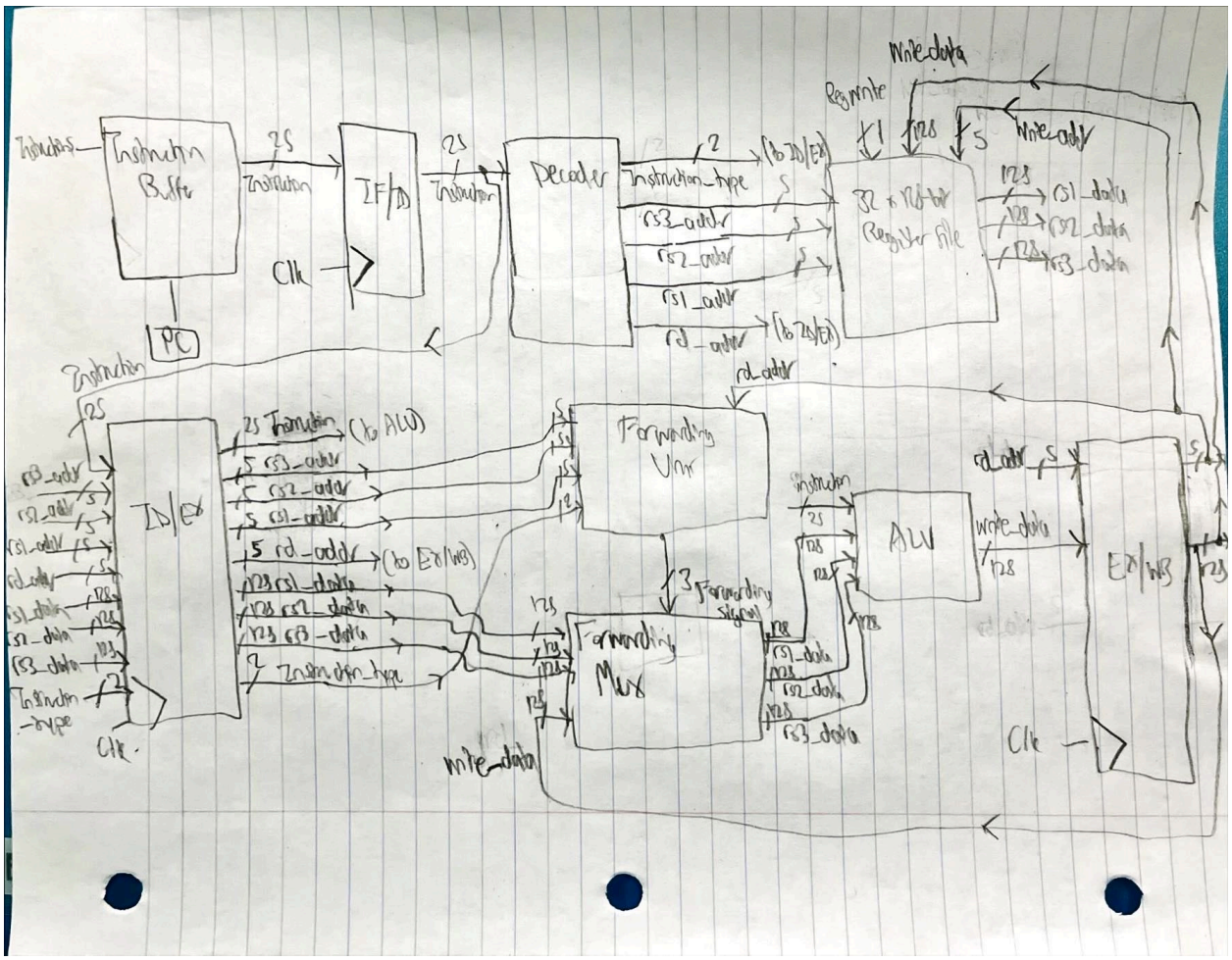
Part 2: All Other Components

Design Procedure

For the second part, we implemented the fetch stage (assembler, instruction buffer, program counter), the decode stage (register file, decoder), the execution stage (forwarding mux, forwarding unit), the write back stage, and the pipeline registers to store any necessary data (IF/ID, ID/EX, EX/WB). A block diagram of our design can be seen below in Figure 32. For reference, register addresses essentially refer to the register numbers.

Figure 31

Block Diagram of Top Level Entity



For the fetch stage, we designed an assembler to convert MIPS assembly language code into a text file of 25-bit binary addresses using C++. The input to the program was a text file with assembly code titled “assembly_instructions.txt” (Figure 32).

Figure 32

Input File to Assembler

```
ldi 15, 0, 5
ldi 18, 0, 4
ldi 19, 0, 2
smaddl 16, 15, 18, 19
nop
cnT1H 18, 15, 16
```

For the assembler code (Figure 33), we used std::unordered_map to map the opcode of each instruction to their instruction syntax in assembly. Then based on the type of instruction read from that line in the input text file, we determined what values to parse next from the line. We converted all integers read from the input file to bits using a helper function.

Figure 33

C++ Code for Assembler

```cpp
// Helper function to convert integer to binary
std::string intToBinary(const std::string& intStr, int numBits) {
    // Convert string to int
    int num;
    try {
        num = std::stoi(intStr);
    }
    catch (const std::exception& e) {
        throw std::invalid_argument("Invalid integer string");
    }

    // Check if numBits is valid
    if (numBits <= 0 || numBits > 32) {
        throw std::invalid_argument("Number of bits must be between 1 and 32");
    }

    // Convert to binary using bitset
    std::bitset<32> bits(num);

    // Get the binary string and take the last numBits
    std::string binaryStr = bits.to_string();
    return binaryStr.substr(32 - numBits);
}


int main(void)
{
        std::ifstream inpf{ "assembly_instructions.txt" };
        std::ofstream outpf{ "instructions.txt" };

    std::unordered_map<std::string, std::string> r4_type_opcodes =
    {
        {"smaddl", "10000"},
        {"smaddh", "10001"},
        {"smsubl", "10010"},
        {"smsubh", "10011"},
        {"lmaddl", "10100"},
        {"lmaddh", "10101"},
        {"lmsubl", "10110"},
        {"lmsubh", "10111"}
    };

    std::unordered_map<std::string, std::string> r3_type_opcodes =
    {
        {"slhi", "1100000001"},
        {"au", "1100000010"},
        {"cntlh", "1100000011"},
        {"ahs", "1100000100"},
        {"and", "1100000101"},
        {"bcw", "1100000110"},
        {"maxws", "1100000111"},
        {"minws", "1100001000"},
        {"mlhu", "1100001001"},
        {"mlhcu", "1100001010"},
        {"or", "1100001011"},
        {"clzh", "1100001100"},
        {"rlh", "1100001101"},
        {"sfwu", "1100001110"},
        {"sfhs", "1100001111"}
    };
```

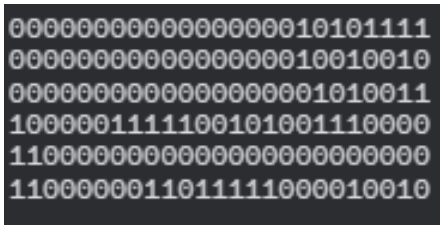
```
if (!inpf)
{
    // Print an error and exit
    std::cerr << "Uh oh, assembly_instructions.txt could not be opened for writing!\n";
    return 1;
}

if (!outpf)
{
    // Print an error and exit
    std::cerr << "Uh oh, instructions.txt could not be opened for writing!\n";
    return 1;
}

std::string instruction;
std::string output;
std::string operation;
while (std::getline(inpf, instruction))
{
    output = "";
    operation = "";
    int i = 0;
    while (instruction[i] != ' ' && instruction[i] != '\0')
    {
        operation += instruction[i++];
    }

    std::cout << "operation: " << operation << '\n';

    // Convert to lower case
    std::transform(operation.begin(), operation.end(), operation.begin(),
        [](unsigned char c) { return std::tolower(c); });

    // Check if load type
    if (operation == "ldi")
    {
        output += "0";

        std::string destination_reg, load_index, immediate;

        // Get destination_reg
        while (instruction[i] != ',')
            destination_reg += instruction[i++];
        i++;

        // Get load_index
        while (instruction[i] != ',')
            load_index += instruction[i++];
        i++;

        // Get immediate
        while (instruction[i] != '\0')
            immediate += instruction[i++];

        output += intToBinary(load_index, 3) + intToBinary(immediate, 16) + intToBinary(destination_reg, 5);

    }
```

```
// Check if r4 instruction
else if (r4_type_opcodes.find(operation) != r4_type_opcodes.end())
{
    // Get r4 opcode
    output += r4_type_opcodes[operation];
    std::string destination_reg, rs3, rs2, rs1;

    // Get destination_reg
    while (instruction[i] != ',')
        destination_reg += instruction[i++];
    i++;

    // Get rs3
    while (instruction[i] != ',')
        rs3 += instruction[i++];
    i++;

    // Get rs2
    while (instruction[i] != ',')
        rs2 += instruction[i++];
    i++;

    // Get rs1
    while (instruction[i] != '\0')
        rs1 += instruction[i++];

    output += intToBinary(rs3, 5) + intToBinary(rs2, 5) + intToBinary(rs1, 5) + intToBinary(destination_reg, 5);
}

// Check if r3 instruction
else if (r3_type_opcodes.find(operation) != r3_type_opcodes.end())
{
    // Get r3 opcode
    output += r3_type_opcodes[operation];
    std::string destination_reg, rs2, rs1;

    // Get destination_reg
    while (instruction[i] != ',')
        destination_reg += instruction[i++];
    i++;

    // Get rs2
    while (instruction[i] != ',')
        rs2 += instruction[i++];
    i++;

    // Get rs1
    while (instruction[i] != '\0')
        rs1 += instruction[i++];

    output += intToBinary(rs2, 5) + intToBinary(rs1, 5) + intToBinary(destination_reg, 5);
}

// Else, we have nop
else
{
    output = "11000000000000000000000000";
}

outpf << output << '\n';
/ End of file reading
```

In Figure 34, the output of the program is shown where each instruction is converted into a 25-bit address that is imputed into the instruction buffer.

Figure 34

Output Text File



The instruction buffer loads 64-25 bit instructions from the text file output of the assembler. On each cycle, the instruction specified by the Program Counter (PC) is fetched, and the value of PC is incremented by 1. In Figure 35, the implementation for the instruction buffer is shown. We make an array of 64 entries with 25 bit instructions and use the program counter as indices in the array. When the write_enable signal is asserted, the *instruction_in* input is written to the instruction buffer. Whenever there is a change in the program counter or in *instruction_in*, the process is run and the instruction being pointed at by the program counter is outputted. The program counter is incremented in the test bench after each cycle.

Figure 35

Instruction Buffer Implementation in VHDL

```
 ----------------------------Instruction Buffer -----------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity instruction_buffer is
    port(
        write_enable: in std_logic;
        program_counter: in integer;
        instruction_in : in std_logic_vector(24 downto 0);
        instruction_out : out std_logic_vector(24 downto 0)
    );

end instruction_buffer;

architecture behavioral of instruction_buffer is
    type instruction_array is array (0 to 63) of std_logic_vector(24 downto 0);
begin
    process(program_counter, instruction_in)
    variable instructions : instruction_array := (others => (others => '0'));
    begin
        if (write_enable = '1') then
            instructions(program_counter) := instruction_in;
        end if;
        instruction_out <= instructions(program_counter);
            -- Increment PC in testbench
    end process;
end behavioral;
```

For the decode stage, we implemented a register file (Figure 36) and a decoder to break down the instruction into components that can be utilized by subsequent pipeline stages, including the ALU. The register file consists of 3 address inputs and 3 data outputs. These correspond to the addresses of rs1, rs2 and rs3 in the 25-bit instruction that is being decoded. The data outputs are the values being stored in these registers. The register file includes a *write_addr* input and a *write_data* input, which specify the destination register (from the EX/WB stage) and the data (from the EX/WB stage) to be written when the (register_write) signal is asserted. In each cycle, three registers are read, and 1 register is written. Similar to the instruction buffer, we make an array of 32 registers where each register has an address (number/index) and data associated with it. Upon assertion of the reset signal, all registers are cleared to zero. For convenience, we maintain that register 0 always holds the value 0 no matter what.

Figure 36

Register File Implementation in VHDL

```
----------------------Register File -------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity register_file is
    port (
        reset       : in  std_logic;
        register_write    : in  std_logic;

        read_addr1  : in  std_logic_vector(4 downto 0);
        read_addr2  : in  std_logic_vector(4 downto 0);
        read_addr3  : in std_logic_vector(4 downto 0);

        write_addr  : in  std_logic_vector(4 downto 0);
        write_data  : in  std_logic_vector(127 downto 0);

        read_data1  : out std_logic_vector(127 downto 0);
        read_data2  : out std_logic_vector(127 downto 0);
        read_data3  : out std_logic_vector(127 downto 0)
    );
end register_file;


architecture Behavioral of register_file is
    type reg_array is array (0 to 31) of std_logic_vector(127 downto 0);
begin
    process(reset, register_write, read_addr1, read_addr2, read_addr3, write_addr, write_data)
      variable registers : reg_array := (others => (others => '0'));
    begin
        if reset = '1' then
            registers := (others => (others => '0'));
            read_data1 <= (others => '0');
            read_data2 <= (others => '0');
            read_data3 <= (others => '0');
        elsif register_write = '1' then
            if write_addr /= "00000" then
                registers(to_integer(unsigned(write_addr))) := write_data;
            end if;
        end if;

        read_data1 <= registers(to_integer(unsigned(read_addr1)));
        read_data2 <= registers(to_integer(unsigned(read_addr2)));
        read_data3 <= registers(to_integer(unsigned(read_addr3)));
    end process;

end Behavioral;
```

In Figure 37, we implemented a decoder by separating the most significant bits of the instruction to determine the type of the instruction. Once we've determined what type of instruction structure we're using, we can account for the use of rs3 (r4 instructions) or immediate like in an *ldi* instruction. If we're not using a register address, we set it equal to 0. However, this is rather trivial as our ALU also deciphers the instruction and only uses the appropriate registers.

Figure 37

Decoder Implementation in VHDL

```
----------------------------Decoder-----------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity decoder is
    port(
        instruction: in std_logic_vector(24 downto 0);
        instruction_type: out std_logic_vector(1 downto 0);
        rs3_addr: out std_logic_vector(4 downto 0);
        rs2_addr: out std_logic_vector(4 downto 0);
        rs1_addr: out std_logic_vector(4 downto 0);
        rd_addr: out std_logic_vector(4 downto 0)
    );

end decoder;

architecture behavioral of decoder is
begin
    process(instruction)
    begin

        -- Figure out instruction type

        -- Load
        if (instruction(24) = '0') then

            -- Set instruction type
            instruction_type <= "00";
            -- Get rs1 and rd
            rs1_addr <= instruction(4 downto 0);
            rd_addr <= instruction(4 downto 0);

            -- Set others to 0
            rs2_addr <= "00000";
            rs3_addr <= "00000";

        -- R4 Instruction
        elsif (instruction(23) = '0') then

            -- Set instruction type
            instruction_type <= "01";

            -- Get rs3, rs2, rs1, rd addresses
            rs3_addr <= instruction(19 downto 15);
            rs2_addr <= instruction(14 downto 10);
            rs1_addr <= instruction(9 downto 5);
            rd_addr <= instruction(4 downto 0);

        -- R3 Instruction Type
        else

            -- Set instruction type
            instruction_type <= "10";

            -- Get rs2, rs1, rd addresses
            rs2_addr <= instruction(14 downto 10);
            rs1_addr <= instruction(9 downto 5);
            rd_addr <= instruction(4 downto 0);

            -- Set rs3 addr to 0
            rs3_addr <= "00000";

        end if;

    end process;
end behavioral;
```

In the execution stage, we designed the forwarding unit and the forwarding multiplexer. The forwarding unit, shown in Figure 38, takes in the instruction type of the instruction to be executed, the addresses of the operands that are being passed into the ALU, and the address of the register that is currently being written to. It outputs a 3-bit vector to determine the input(s) that data needs to be forwarded to. The forwarding unit does the necessary address comparisons based on the instruction_type of the instruction to be executed, and then sets the corresponding flag in the output *forward* vector, where the most significant bit of the vector represents the forwarding flag for rs3. This 3-bit output signal then goes to the forwarding multiplexer.

Figure 38

Forwarding Unit Implementation in VHDL

```
----------------------------Forwarding Unit-----------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity forwarding_unit is
    port(
        rs1_addr: in std_logic_vector(4 downto 0);
        rs2_addr: in std_logic_vector(4 downto 0);
        rs3_addr: in std_logic_vector(4 downto 0);
        rd_addr: in std_logic_vector(4 downto 0);
        instruction_type: in std_logic_vector(1 downto 0);
        forward: out std_logic_vector(2 downto 0)
    );
end forwarding_unit;

architecture behavioral of forwarding_unit is
begin
    process(rs1_addr, rs2_addr, rs3_addr, rd_addr, instruction_type)
    begin

        forward <= "000";
        -- Figure out instruction type

        -- Load
        if (instruction_type = "00") then
            if (rd_addr = rs1_addr) then
                forward(0) <= '1';
            end if;

        -- R4
        elsif (instruction_type = "01") then

            if (rd_addr = rs1_addr) then
                forward(0) <= '1';
            end if;

            if (rd_addr = rs2_addr) then
                forward(1) <= '1';
            end if;

            if (rd_addr = rs3_addr) then
                forward(2) <= '1';
            end if;


        -- R3
        else

            if (rd_addr = rs1_addr) then
                forward(0) <= '1';
            end if;

            if (rd_addr = rs2_addr) then
                forward(1) <= '1';
            end if;

        end if;

    end process;
end behavioral;
```

The forwarding multiplexer ensures the most recent data values are used during instruction execution to handle data hazards. The *forwarding_mux* entity takes three register inputs (*rs1_in*, *rs2_in*, *rs3_in*), the data to be forwarded (*forwarded_data*), and a forwarding signal from the forwarding unit that tells us which inputs we need to forward data to (*forwarding_signal*). The process assigns the inputs to the outputs (rs1_out, rs2_out, rs3_out). If the forwarded_signal flags are set, the corresponding register output is updated as well. For example, if *forwarding_signal (2)* is set, then *rs3_out* takes the value of *forwarded_data* (coming from the EX/WB register) instead of *rs3_in* (coming from ID/EX register) .

Figure 39

Forwarding Mux Implementation in VHDL

```
--------------------------------Forwarding Mux-----------------------------------

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

entity forwarding_mux is
    port(
        rs1_in: in std_logic_vector(127 downto 0);
        rs2_in: in std_logic_vector(127 downto 0);
        rs3_in: in std_logic_vector(127 downto 0);
        forwarded_data: in std_logic_vector(127 downto 0);
        forwarding_signal: in std_logic_vector(2 downto 0);
        rs1_out: out std_logic_vector(127 downto 0);
        rs2_out: out std_logic_vector(127 downto 0);
        rs3_out: out std_logic_vector(127 downto 0)
    );
end forwarding_mux;

architecture behavioral of forwarding_mux is
begin
    process(rs1_in, rs2_in, rs3_in, forwarded_data, forwarding_signal)
    begin
        rs1_out <= rs1_in;
        rs2_out <= rs2_in;
        rs3_out <= rs3_in;

        if (forwarding_signal(2) = '1') then
            rs3_out <= forwarded_data;
        end if;

        if (forwarding_signal(1) = '1') then
            rs2_out <= forwarded_data;
        end if;

        if (forwarding_signal(0) = '1') then
            rs1_out <= forwarded_data;
        end if;

    end process;
end behavioral;
```

To make sure we maintain the necessary data to complete each stage of the pipeline, we implemented the pipeline registers to pass along data corresponding to each instruction. In Figure 40, we can see the implementation of our IF/ID register which stores the instruction that was fetched.

Figure 40

IF/ID Pipeline Register in VHDL

```
-------------------------------- IF/ID ----------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity if_id_reg is
    port(
        instruction_d : in std_logic_vector(24 downto 0);
        clk : in std_logic;
        instruction_q : out std_logic_vector(24 downto 0)
    );

end if_id_reg;

architecture behavioral of if_id_reg is
begin
    process(clk)
    begin
        if rising_edge(clk) then
            instruction_q <= instruction_d;
        end if;
    end process;
end behavioral;
```

In Figure 41, we can see the implementation of the ID/EX register. This passes along the instruction that was decoded, its instruction_type, the register addresses of its operands, the data of the operands, and the destination register. The register addresses of the operands and the instruction type get passed along so that they can be used to determine if we need to forward data or not. The destination register is also passed along for this reason, and so that we know where to write our data back to.

Figure 41

ID/EX Pipeline Register in VHDL

```
------------------------------- ID/EX ---------------------------------

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

entity id_ex_reg is
    port(

    -- Inputs
    clk : in std_logic;

    instruction_d : in std_logic_vector(24 downto 0);
    rs1_addr_d: in std_logic_vector(4 downto 0);
    rs2_addr_d: in std_logic_vector(4 downto 0);
    rs3_addr_d: in std_logic_vector(4 downto 0);
    rsd_addr_d: in std_logic_vector(4 downto 0);
    instruction_type_d: in std_logic_vector(1 downto 0);
    rs1_data_d: in std_logic_vector(127 downto 0);
    rs2_data_d: in std_logic_vector(127 downto 0);
    rs3_data_d: in std_logic_vector(127 downto 0);

    -- Outputs
    instruction_q : out std_logic_vector(24 downto 0);
    rs1_addr_q: out std_logic_vector(4 downto 0);
    rs2_addr_q: out std_logic_vector(4 downto 0);
    rs3_addr_q: out std_logic_vector(4 downto 0);
    rsd_addr_q: out std_logic_vector(4 downto 0);
    instruction_type_q: out std_logic_vector(1 downto 0);
    rs1_data_q: out std_logic_vector(127 downto 0);
    rs2_data_q: out std_logic_vector(127 downto 0);
    rs3_data_q: out std_logic_vector(127 downto 0)

    );

end id_ex_reg;

architecture behavioral of id_ex_reg is
begin
    process(clk)
    begin
        if rising_edge(clk) then
            instruction_q <= instruction_d;
            rs1_addr_q <= rs1_addr_d;
            rs2_addr_q <= rs2_addr_d;
            rs3_addr_q <= rs3_addr_d;
            rsd_addr_q <= rsd_addr_d;
            instruction_type_q <= instruction_type_d;
            rs1_data_q <= rs1_data_d;
            rs2_data_q <= rs2_data_d;
            rs3_data_q <= rs3_data_d;
        end if;
    end process;
end behavioral;
```

Figure 42 shows our implementation of the EX/WB register which stores the address of the register that is being written to in the write back stage, and the data that is going to be written. The address stored in the EX/WB register is also used to determine if we need to forward data to the execution stage.

Figure 42

EX/WB Pipeline Register in VHDL

```
------------------------------ EX/WB ---------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ex_wb_reg is
    port(
        clk : in std_logic;
        rd_addr_d : in std_logic_vector(4 downto 0);
        rd_data_d : in std_logic_vector(127 downto 0);

        rd_addr_q : out std_logic_vector(4 downto 0);
        rd_data_q : out std_logic_vector(127 downto 0)
    );

end ex_wb_reg;

architecture behavioral of ex_wb_reg is
begin
    process(clk)
    begin
        if rising_edge(clk) then
            rd_addr_q <= rd_addr_d;
            rd_data_q <= rd_data_d;
        end if;
    end process;
end behavioral;
```

After designing each component, we created our final top-level multimedia unit using a structural design approach, mapping everything together using variables in VHDL. We also added a process that would write some of the contents of each pipeline stage to an output file called *results.txt* on every clock edge. This allowed us to verify that our overall design was working correctly, ensuring that the pipelining and forwarding was working as intended.

Figure 43

Top-Level Implementation

```
----------------------------------------------------------------------------
--
-- Description : Top-Level Entity
--
----------------------------------------------------------------------------

--{{ Section below this comment is automatically maintained
--    and may be overwritten
--{entity {multimedia_alu} architecture {behavioral}}


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use work.all;

entity multimedia_unit is
    port(
        write_enable : in std_logic;
        reset: in std_logic;
        clk : in std_logic;
        program_counter: in integer;
        instruction_in : in std_logic_vector(24 downto 0)

    );
end multimedia_unit;

architecture structural of multimedia_unit is

function to_hstring (SLV : signed) return string is
  variable L : LINE;
begin
  hwrite(L, std_logic_vector(SLV));
  return L.all;
end function to_hstring;


signal instruction_stage1 : std_logic_vector(24 downto 0);
signal instruction_stage2 : std_logic_vector(24 downto 0);
signal instruction_stage3 : std_logic_vector(24 downto 0);

signal instruction_type_stage2 : std_logic_vector(1 downto 0);
signal rs3_addr_stage2 : std_logic_vector(4 downto 0);
signal rs2_addr_stage2 : std_logic_vector(4 downto 0);
signal rs1_addr_stage2 : std_logic_vector(4 downto 0);
signal rd_addr_stage2  : std_logic_vector(4 downto 0);

signal rs1_data_stage2 : std_logic_vector(127 downto 0);
signal rs2_data_stage2 : std_logic_vector(127 downto 0);
signal rs3_data_stage2 : std_logic_vector(127 downto 0);

signal instruction_type_stage3 : std_logic_vector(1 downto 0);
signal rs3_addr_stage3 : std_logic_vector(4 downto 0);
signal rs2_addr_stage3 : std_logic_vector(4 downto 0);
signal rs1_addr_stage3 : std_logic_vector(4 downto 0);
signal rd_addr_stage3  : std_logic_vector(4 downto 0);

signal rs1_data_stage3 : std_logic_vector(127 downto 0);
signal rs2_data_stage3 : std_logic_vector(127 downto 0);
signal rs3_data_stage3 : std_logic_vector(127 downto 0);

signal rs1_data_stage3_2 : std_logic_vector(127 downto 0);
signal rs2_data_stage3_2 : std_logic_vector(127 downto 0);
signal rs3_data_stage3_2 : std_logic_vector(127 downto 0);

signal forwarding_signal : std_logic_vector(2 downto 0);

signal write_data_stage3 : bit_vector(127 downto 0);
```

```
signal rd_addr_stage4  : std_logic_vector(4 downto 0);
signal write_data_stage4 : std_logic_vector(127 downto 0);


begin

    instr_buffer: entity instruction_buffer port map(
        write_enable => write_enable,
        program_counter => program_counter,
        instruction_in => instruction_in,

        instruction_out => instruction_stage1
        );

    if_id: entity if_id_reg port map(
        instruction_d => instruction_stage1,
        clk => clk,

        instruction_q => instruction_stage2
        );

    decoder: entity decoder port map(
        instruction => instruction_stage2,

        instruction_type => instruction_type_stage2,
        rs3_addr => rs3_addr_stage2,
        rs2_addr => rs2_addr_stage2,
        rs1_addr => rs1_addr_stage2,
        rd_addr => rd_addr_stage2
        );

    reg_file: entity register_file port map(
        reset => reset,
        register_write => '1',
        read_addr1 => rs1_addr_stage2,
        read_addr2 => rs2_addr_stage2,
        read_addr3 => rs3_addr_stage2,
        write_addr => rd_addr_stage4,
        write_data => write_data_stage4,

        read_data1 => rs1_data_stage2,
        read_data2 => rs2_data_stage2,
        read_data3 => rs3_data_stage2
        );

    id_ex: entity id_ex_reg port map(
        clk => clk,
        instruction_d => instruction_stage2,
        rs1_addr_d => rs1_addr_stage2,
        rs2_addr_d => rs2_addr_stage2,
        rs3_addr_d => rs3_addr_stage2,
        rsd_addr_d => rd_addr_stage2,
        instruction_type_d => instruction_type_stage2,
        rs1_data_d => rs1_data_stage2,
        rs2_data_d => rs2_data_stage2,
        rs3_data_d => rs3_data_stage2,

        instruction_q => instruction_stage3,
        rs1_addr_q => rs1_addr_stage3,
        rs2_addr_q => rs2_addr_stage3,
        rs3_addr_q => rs3_addr_stage3,
        rsd_addr_q => rd_addr_stage3,
        instruction_type_q => instruction_type_stage3,
        rs1_data_q => rs1_data_stage3,
        rs2_data_q => rs2_data_stage3,
        rs3_data_q => rs3_data_stage3
        );
```

```
    forwarding_unit: entity forwarding_unit port map(
        rs1_addr => rs1_addr_stage3,
        rs2_addr => rs2_addr_stage3,
        rs3_addr => rs3_addr_stage3,
        rd_addr => rd_addr_stage4,
        instruction_type => instruction_type_stage3,

        forward => forwarding_signal
        );

    forwarding_mux: entity forwarding_mux port map(
        rs1_in => rs1_data_stage3,
        rs2_in => rs2_data_stage3,
        rs3_in => rs3_data_stage3,
        forwarded_data => write_data_stage4,
        forwarding_signal => forwarding_signal,

        rs1_out => rs1_data_stage3_2,
        rs2_out => rs2_data_stage3_2,
        rs3_out => rs3_data_stage3_2
        );

    alu: entity multimedia_alu port map(
        instruction => (to_bitvector(instruction_stage3)),
        rs1 => (to_bitvector(rs1_data_stage3_2)),
        rs2 => (to_bitvector(rs2_data_stage3_2)),
        rs3 => (to_bitvector(rs3_data_stage3_2)),

        rd => write_data_stage3
        );

    ex_wb: entity ex_wb_reg port map(
        clk => clk,
        rd_addr_d => rd_addr_stage3,
        rd_data_d => to_stdlogicvector(write_data_stage3),

        rd_addr_q => rd_addr_stage4,
        rd_data_q => write_data_stage4
        );

    process(clk)
        file output_file : text open write_mode is "results.txt";
        variable line_out : line;

        variable instruction_fetch_stage : string(1 to 47);
        variable instruction_decode_stage : string(1 to 47);
        variable instruction_decode_stage2 : string(1 to 45);
        variable instruction_execute_stage: string(1 to 48);
        variable instruction_execute_stage2: string(1 to 59);
        variable instruction_execute_stage3: string(1 to 39);
        variable instruction_writeback_stage : string(1 to 67);
    begin
        if rising_edge(clk) then
        instruction_fetch_stage := "Fetching instruction: " & to_string(instruction_stage1);
        write(line_out, instruction_fetch_stage);
        writeline(output_file, line_out);

        instruction_decode_stage := "Decoding instruction: " & to_string(instruction_stage2);
        write(line_out, instruction_decode_stage);
        writeline(output_file, line_out);

        instruction_decode_stage2 := "rs3: " & to_string(rs3_addr_stage2) & ", rs2: " & to_string(rs2_addr_stage2) & ", rs1: " & to_string(rs1_addr_stage2) & ", rd: " & to_string(rd_addr_stage2);
        write(line_out, instruction_decode_stage2);
        writeline(output_file, line_out);

        instruction_execute_stage := "Executing Instruction: " & to_string(instruction_stage3);
        write(line_out, instruction_execute_stage);
        writeline(output_file, line_out);



        instruction_execute_stage2 := "rs3: " & to_string(rs3_addr_stage3) & ", rs2: " & to_string(rs2_addr_stage3) & ", rs1: " & to_string(rs1_addr_stage3) & ", rd (of WB stage): " & to_string(rd_addr_stage4);
        write(line_out, instruction_execute_stage2);
        writeline(output_file, line_out);


        instruction_execute_stage3 := "Forwarding Signal (rs3, rs2, rs1) = " & to_string(forwarding_signal);
        write(line_out, instruction_execute_stage3);
        writeline(output_file, line_out);

        instruction_writeback_stage := "Writing register " & to_string(rd_addr_stage4) & " with value: " & to_hstring(write_data_stage4);
        write(line_out, instruction_writeback_stage);
        writeline(output_file, line_out);

        write(line_out, LF);
        writeline(output_file, line_out);

        write(line_out, LF);
        writeline(output_file, line_out);


        end if;
    end process;

end structural;
```

Testing Our Results:

We first tested each individual entity that we built before testing the final design. Figure 44 shows the simulation waveform of testing the instruction buffer. Here, we made sure that the value coming out of the instruction buffer matched instruction stored at the location pointed at by the PC.

Figure 44

Instruction Buffer Waveform

Figure 45 shows the simulation waveform of testing the register file. Here we made sure that the correct registers were being read, and the correct registers were being written to when the *register_write* was asserted.

Figure 45

Register File Waveform

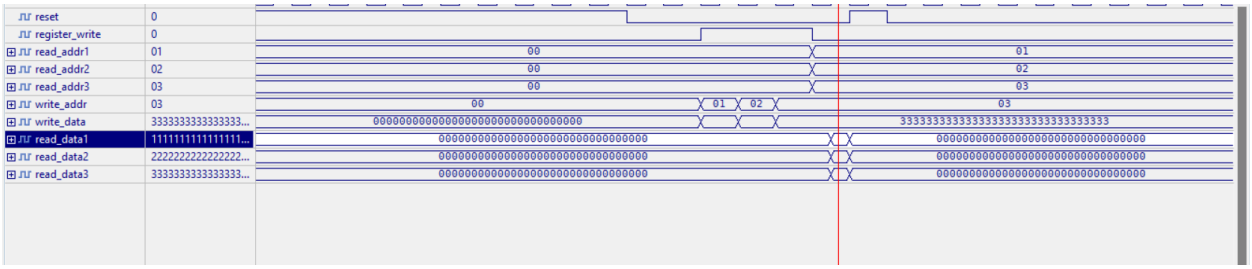


Figure 46 shows the simulation waveform of testing our decoder. Here, we passed in the bits for a sample r3 instruction and made sure the right register numbers were being outputted.

Figure 46

Decoder Waveform

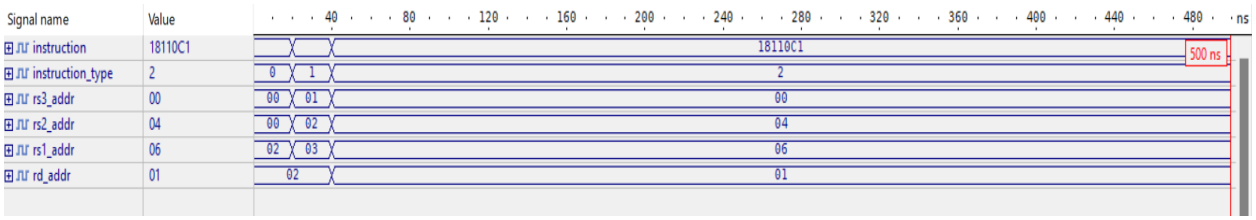


Figure 47 shows the simulation waveform of testing our forwarding-unit. Here, we passed in the bits for each type of instruction and made sure that the destination register (rd) updated to the correct value depending on the instruction type.

Figure 47

Forwarding-Unit Waveform

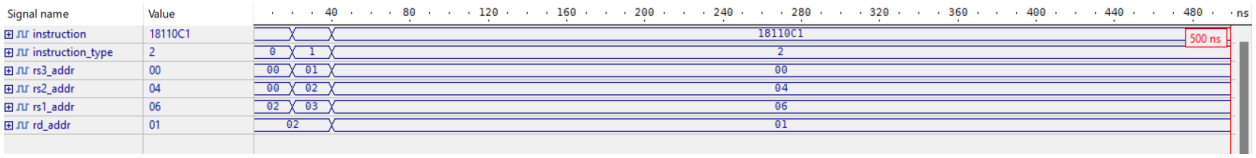
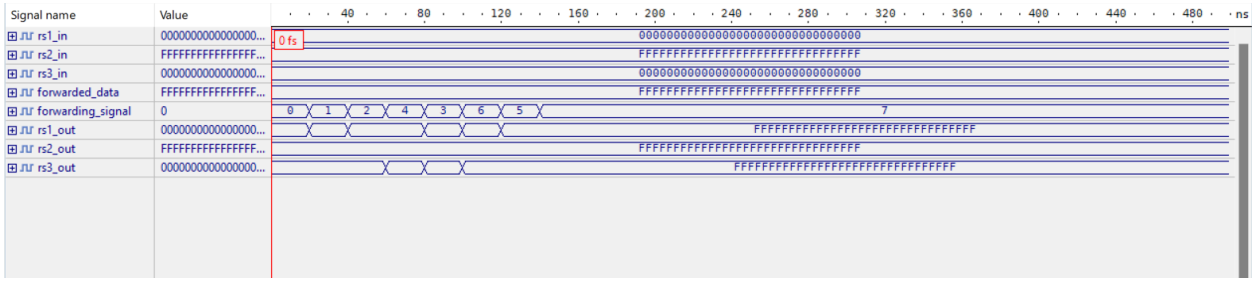


Figure 48 shows the simulation waveform of testing our forwarding mux. Here, we passed in 3 register inputs (rs1_in, rs2_in, rs3_in) and forwarding data with a forwarding signal. As the forwarding signal changes we can see the forwarding data updating the register output corresponding to the forward signal.

Figure 48

Forwarding-Mux Waveform



We then tested the pipeline registers and made sure they updated their outputs on rising edges of the clock. Finally, we tested our top level entity and verified our pipelining was working correctly. The waveform of the register contents can be seen below in Figure 49. The contents written to the *results.txt* file can be shown in Figure 50. From our file, we can see that the pipeline first fills up in the first four cycles. In the fourth cycle, we write a value of 5 into register 15. In the following cycle, we write a value of 4 into register 18. Then, in the 6th cycle, we write a value of 2 into register 19. While we are doing that, we also execute a multiply-add low instruction with registers 15, 18, 19 (representing rs3, rs2, and rs1, respectively), and the destination register being register 16. Here, we are able to correctly detect the need to forward the newer value of 2 to replace the old value of register 19 that is being passed into the ALU. Hence, we end up properly multiplying 5 by 4, and then adding 2 to that. This is why a value of 22 (0x16) is written to register 16 during the 7th cycle.

Figure 49

Top-Level Entity Waveform

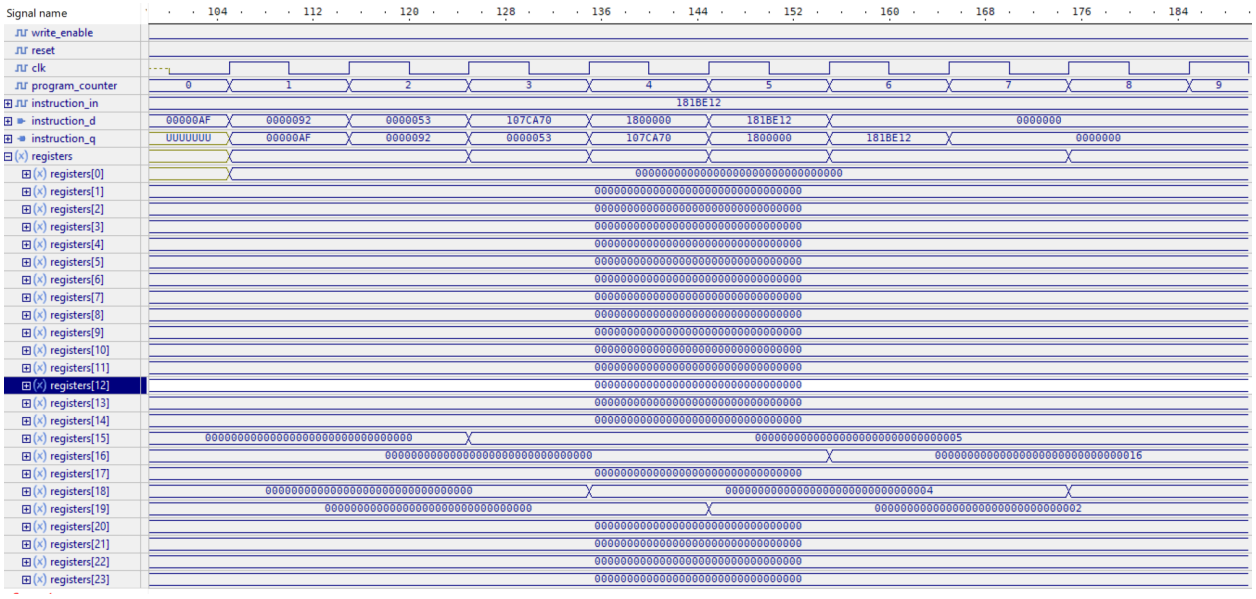


Figure 50

Results File

```
Fetching instruction: 00000000000000000010101111
Decoding instruction: UUUUUUUUUUUUUUUUUUUUUUUUUU
rs3: 00000, rs2: UUUUU, rs1: UUUUU, rd: UUUUU
Executing Instruction: UUUUUUUUUUUUUUUUUUUUUUUUUU
rs3: UUUUU, rs2: UUUUU, rs1: UUUUU, rd (of WB stage): UUUUU
Forwarding Signal (rs3, rs2, rs1) = 011
Writing register UUUUU with value: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX



Fetching instruction: 00000000000000000010010010
Decoding instruction: 00000000000000000010101111
rs3: 00000, rs2: 00000, rs1: 01111, rd: 01111
Executing Instruction: UUUUUUUUUUUUUUUUUUUUUUUUUU
rs3: 00000, rs2: UUUUU, rs1: UUUUU, rd (of WB stage): UUUUU
Forwarding Signal (rs3, rs2, rs1) = 011
Writing register UUUUU with value: 00000000000000000000000000000000



Fetching instruction: 00000000000000000001010011
Decoding instruction: 00000000000000000010010010
rs3: 00000, rs2: 00000, rs1: 10010, rd: 10010
Executing Instruction: 00000000000000000010101111
rs3: 00000, rs2: 00000, rs1: 01111, rd (of WB stage): UUUUU
Forwarding Signal (rs3, rs2, rs1) = 000
Writing register UUUUU with value: 00000000000000000000000000000000



Fetching instruction: 10000011111001010011110000
Decoding instruction: 00000000000000000001010011
rs3: 00000, rs2: 00000, rs1: 10011, rd: 10011
Executing Instruction: 00000000000000000010010010
rs3: 00000, rs2: 00000, rs1: 10010, rd (of WB stage): 01111
Forwarding Signal (rs3, rs2, rs1) = 000
Writing register 01111 with value: 00000000000000000000000000000005



Fetching instruction: 11000000000000000000000000
Decoding instruction: 10000011111001010011110000
rs3: 01111, rs2: 10010, rs1: 10011, rd: 10000
Executing Instruction: 00000000000000000001010011
rs3: 00000, rs2: 00000, rs1: 10011, rd (of WB stage): 10010
Forwarding Signal (rs3, rs2, rs1) = 000
Writing register 10010 with value: 00000000000000000000000000000004
```

```
Fetching instruction: 11000000110111111000010010
Decoding instruction: 11000000000000000000000000
rs3: 00000, rs2: 00000, rs1: 00000, rd: 00000
Executing Instruction: 10000011111001010011110000
rs3: 01111, rs2: 10010, rs1: 10011, rd (of WB stage): 10011
Forwarding Signal (rs3, rs2, rs1) = 001
Writing register 10011 with value: 00000000000000000000000000000002



Fetching instruction: 00000000000000000000000000
Decoding instruction: 11000000110111111000010010
rs3: 00000, rs2: 01111, rs1: 10000, rd: 10010
Executing Instruction: 11000000000000000000000000
rs3: 00000, rs2: 00000, rs1: 00000, rd (of WB stage): 10000
Forwarding Signal (rs3, rs2, rs1) = 000
Writing register 10000 with value: 00000000000000000000000000000016



Fetching instruction: 00000000000000000000000000
Decoding instruction: 00000000000000000000000000
rs3: 00000, rs2: 00000, rs1: 00000, rd: 00000
Executing Instruction: 11000000110111111000010010
rs3: 00000, rs2: 01111, rs1: 10000, rd (of WB stage): 00000
Forwarding Signal (rs3, rs2, rs1) = 000
Writing register 00000 with value: 00000000000000000000000000000016



Fetching instruction: 00000000000000000000000000
Decoding instruction: 00000000000000000000000000
rs3: 00000, rs2: 00000, rs1: 00000, rd: 00000
Executing Instruction: 00000000000000000000000000
rs3: 00000, rs2: 00000, rs1: 00000, rd (of WB stage): 10010
Forwarding Signal (rs3, rs2, rs1) = 000
Writing register 10010 with value: 00000000000000000000000000000003
```

Conclusion

In Part 1 of the project, we successfully developed and verified the VHDL source code for the multimedia ALU functions at the third pipeline stage. This involved implementing various SIMD operations, including arithmetic, logical, and shift operations, using a behavioral approach in VHDL. The verification process was carried out using simulation tools to ensure that each function performed as expected.

In Part 2 of the project, we successfully designed and verified our components for the 4 stage pipelined multimedia unit. This involved implementing the fetch stage (assembler, instruction buffer, program counter), the decode stage (register file, decoder), the execution stage (forwarding mux, forwarding unit), the write back stage, and the pipeline registers to store any necessary data (IF/ID, ID/EX, EX/WB).

Throughout the entire process, we encountered several challenges. For example, implementing saturation arithmetic for signed operations was particularly challenging as we had to ensure that the results of arithmetic operations did not exceed the defined limits for 32-bit and 64-bit values, and this required additional logic for comparison and boundary checks. We also found that many of the arithmetic operations required a lot of tedious type conversion where we had to convert between the bit_vector type and signed/unsigned vector types. Lastly, verifying the correctness of each operation involved careful testing with different input values. Reading the inputs on the waveform was difficult since we were working with a large number of bits at once. Despite these challenges, we were able to achieve successful verification of all implemented ALU functions. In the second part, verifying each component became very challenging since each entity required its own testbench. This worked in our favor though since we were able to verify all our components before we reached the top-level which made verifying our final result much easier. Going through the struggles of verification early on made us more confident that our final design would function correctly.