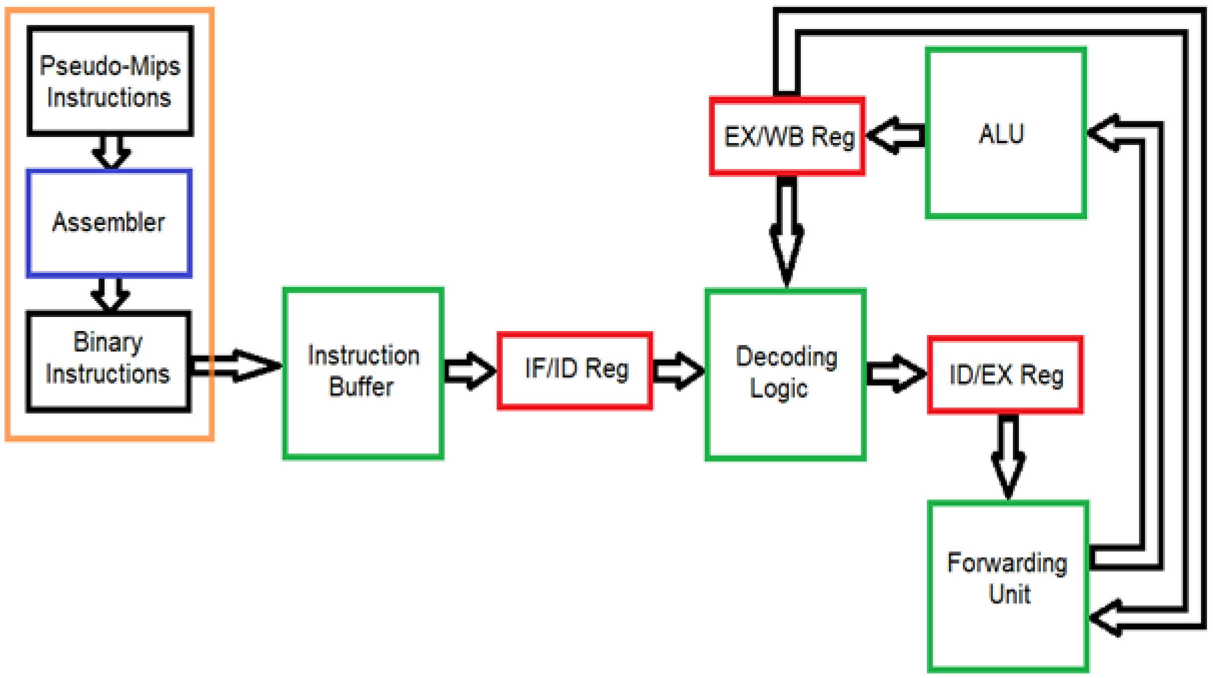
Goals

The purpose of this project was to implement a structural and behavioral design of a four-stage pipelined multimedia unit with a reduced set of multimedia instructions. The popeline is similar to those in the Sony Cell SPU and Intel SSE architectures. The pipeline was developed in a structural/RTL manner with several modules operating simultaneously. Each module represents a pipelined stage with its interstage register. The components incorporated were an ALU, control unit, forwarding unit, instruction buffer and various intermediate registers. These components fit together in a pipeline, that is a chain of tasks that allow for multiple instructions to be worked on in an assembly line fashion. These instructions have been written and coded in a pseudo-mips manner so that the pipeline and it's components can operate in unison. The instructions are read in the instruction buffer, and passed through to the ALU, where they are logically executed. The forwarding unit is able to catch and assist with hazards that certain instructions may bring up. All throughout are interstate registers, that allow for the system to "pulse" on a clock signal, allowing for each stage to be executed and then have it's data passed on to the next. The end results are able to be compared to our expected results, generated by the testbench. When all of the components are brought together under the Pipeline toplevel, we are able to see the flow of data through each cycle of execution and how the various registers are manipulated along the way.

Block Diagram

The overall function of the multimedia unit can be expressed through the following diagram:



Key: External (Assembler), Register, Pipeline Unit

Following the diagram, we convert pseudo-mips instructions into 25-bit binary instructions in the assembler. These instructions then pass to the instruction buffer, where they are primed in the pipeline. Passing through the IF/ID register, they reach the decoding logic, where the instruction type and necessary register names are retrieved. Next in the line is the forwarding unit, which is capable of retrieving data that has already been calculated for use and also sends the necessary information to the ALU for calculation. After calculations are made, the final data is stored in the EX/WB register, and made available to the pipeline again by inserting it back into the loop.

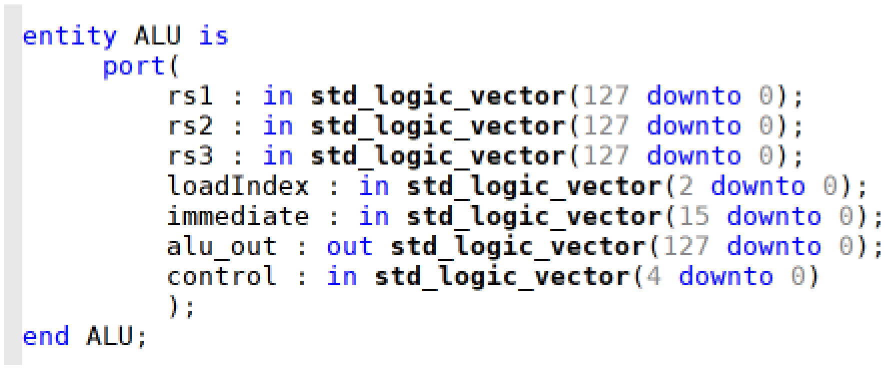
Design Procedure

The objective was to create a more efficient multimedia unit design than the single cycle processor. This pipeline needs just the clock signal to start the process. The interstate registers are designed to carry data between modules in the datapath. Each is clocked so that multiple instructions can be carried out at once without altering the datapath modules. The only data needed to be passed throughout the pipeline is the 25-bit instruction code, registers, and control bits. Interstate Registers dissect the instruction to give the datapath all the information needed to execute, but doesn't pass unnecessary data.

Functionality

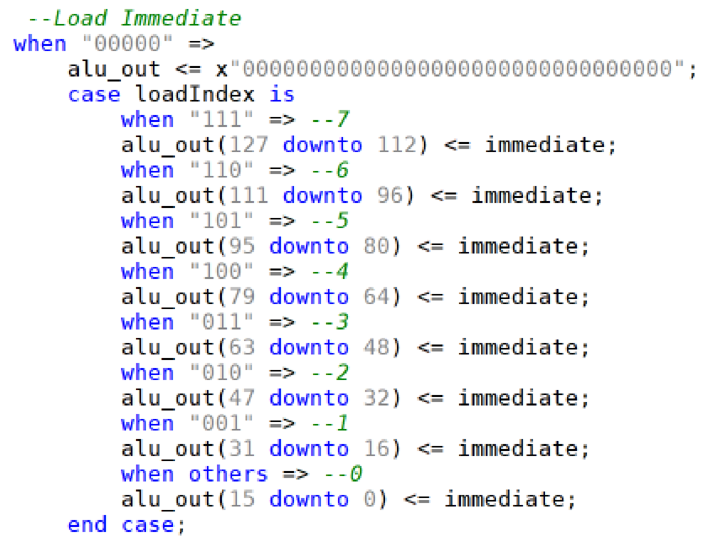
ALU

The ALU takes up to three inputs from the register file and calculates the result based on the current instruction to be performed. As seen below by the port declarations you can see rs1,rs2 and rs3 as the three inputs from the register file.

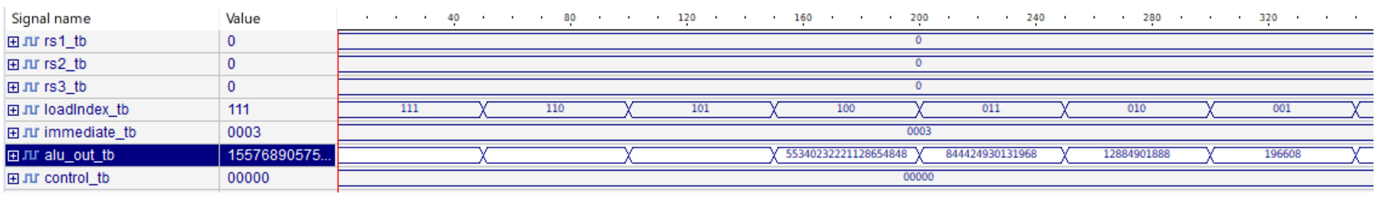


```
entity ALU is
     port(
         rs1 : in std_logic_vector(127 downto 0);
         rs2 : in std_logic_vector(127 downto 0);
         rs3 : in std_logic_vector(127 downto 0);
         loadIndex : in std_logic_vector(2 downto 0);
         immediate : in std_logic_vector(15 downto 0);
         alu_out : out std_logic_vector(127 downto 0);
         control : in std_logic_vector(4 downto 0)
         );
end ALU;
```

When control has a value of 00000 we load a 16-bit immediate value from the instruction field into a 16 bit field specified by the Load Index. This can be seen below where the value for immediate is determined by the Load Index.

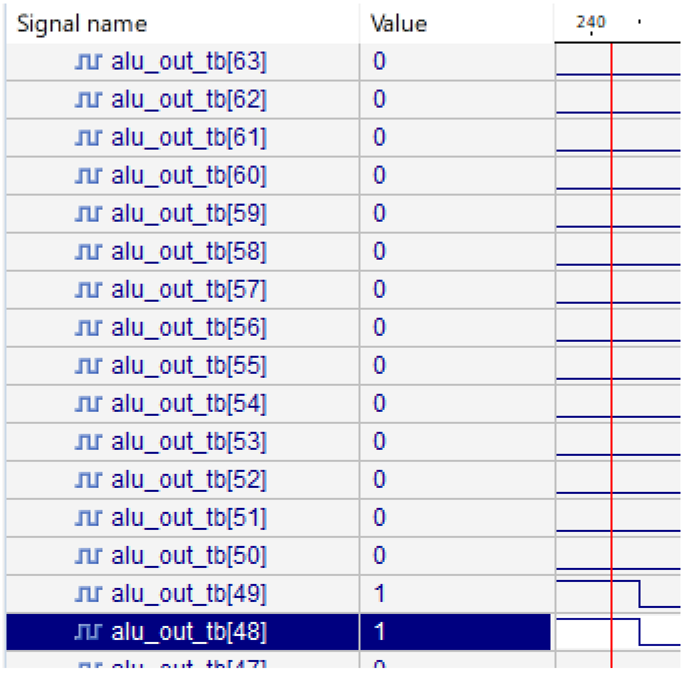


```
  --Load Immediate
 when "00000" =>
     alu_out <= x"00000000000000000000000000000000";
     case loadIndex is
         when "111" => --7
         alu_out(127 downto 112) <= immediate;
         when "110" => --6
         alu_out(111 downto 96) <= immediate;
         when "101" => --5
         alu_out(95 downto 80) <= immediate;
         when "100" => --4
         alu_out(79 downto 64) <= immediate;
         when "011" => --3
         alu_out(63 downto 48) <= immediate;
         when "010" => --2
         alu_out(47 downto 32) <= immediate;
         when "001" => --1
         alu_out(31 downto 16) <= immediate;
         when others => --0
         alu_out(15 downto 0) <= immediate;
     end case;
```



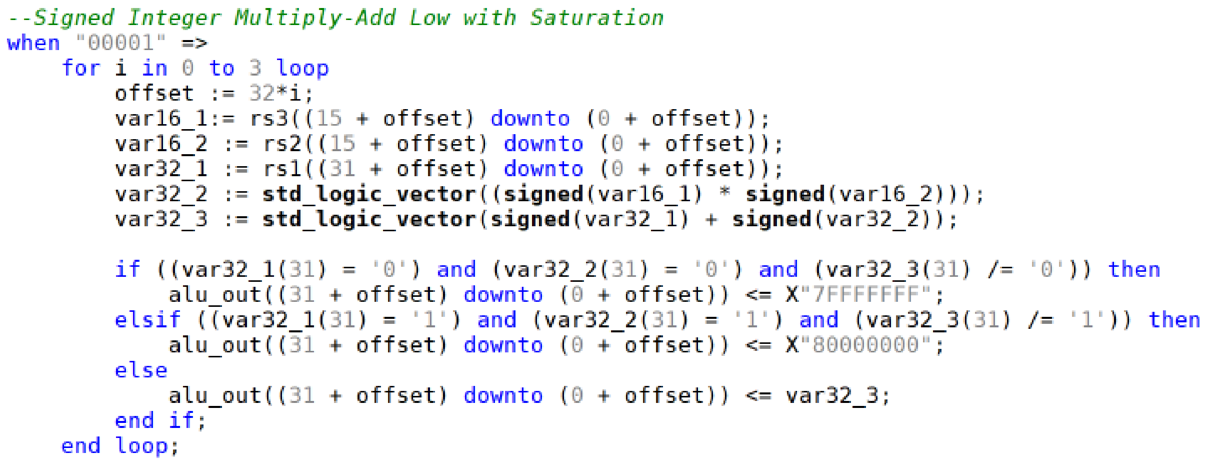
```
Signal name      Value
rs1_tb           0
rs2_tb           0
rs3_tb           0
loadIndex_tb     111
immediate_tb     0003
alu_out_tb       15576890575...
control_tb       00000
```

Here we verify the Load Immediate where we load a 16-bit Immediate value (0000000000000011) into rd from index 63 downto 48

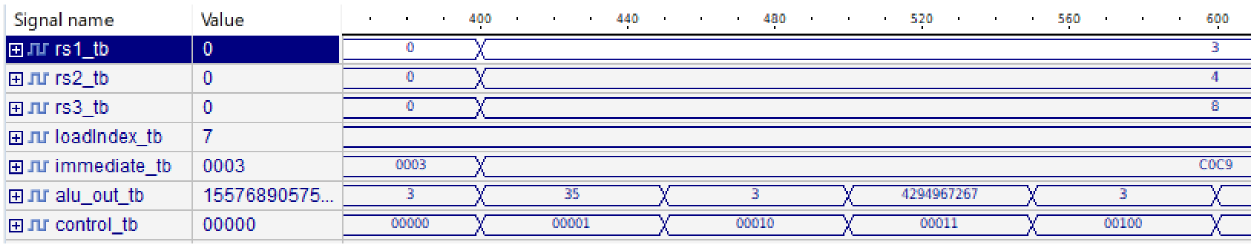


```
Signal name        Value
alu_out_tb[63]     0
alu_out_tb[62]     0
alu_out_tb[61]     0
alu_out_tb[60]     0
alu_out_tb[59]     0
alu_out_tb[58]     0
alu_out_tb[57]     0
alu_out_tb[56]     0
alu_out_tb[55]     0
alu_out_tb[54]     0
alu_out_tb[53]     0
alu_out_tb[52]     0
alu_out_tb[51]     0
alu_out_tb[50]     0
alu_out_tb[49]     1
alu_out_tb[48]     1
alu_out_tb[47]     0
```

The instruction to be performed is dependent on the value given by the input control. The inputs from the register file are used to calculate the result which is outputted to alu_out. Here we verify the instruction Signed Integer Multiple-Add Low with Saturation when control has a value of 00001. We multiply rs3 and rs2, then add the products to rs1 and output the result. The values can be verified by checking the decimal values, for rs1 = 3, rs2 = 4, rs3 = 8. The output for this operation should be 35 and this is verified by the waveform.

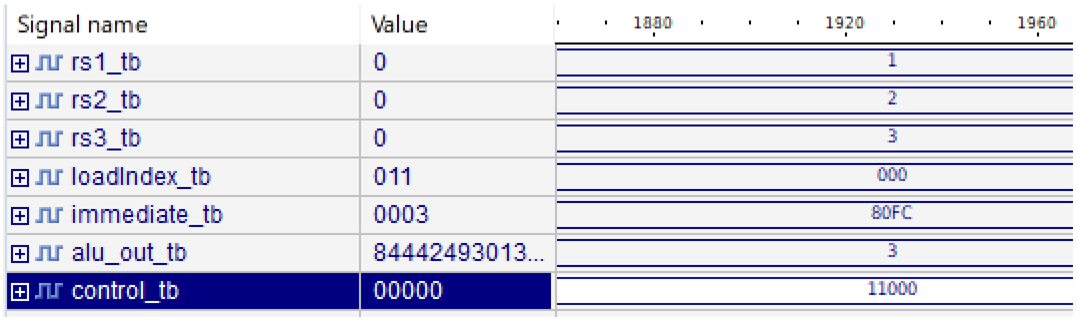


```
--Signed Integer Multiply-Add Low with Saturation
when "00001" =>
    for i in 0 to 3 loop
        offset := 32*i;
        var16_1:= rs3((15 + offset) downto (0 + offset));
        var16_2 := rs2((15 + offset) downto (0 + offset));
        var32_1 := rs1((31 + offset) downto (0 + offset));
        var32_2 := std_logic_vector((signed(var16_1) * signed(var16_2)));
        var32_3 := std_logic_vector(signed(var32_1) + signed(var32_2));

        if ((var32_1(31) = '0') and (var32_2(31) = '0') and (var32_3(31) /= '0')) then
            alu_out((31 + offset) downto (0 + offset)) <= X"7FFFFFFF";
        elsif ((var32_1(31) = '1') and (var32_2(31) = '1') and (var32_3(31) /= '1')) then
            alu_out((31 + offset) downto (0 + offset)) <= X"80000000";
        else
            alu_out((31 + offset) downto (0 + offset)) <= var32_3;
        end if;
    end loop;
```



```
Signal name      Value          
rs1_tb           0              0 | 3
rs2_tb           0              0 | 4
rs3_tb           0              0 | 8
loadIndex_tb     7              
immediate_tb     0003           0003 | C0C9
alu_out_tb       15576890575... 3 | 35 | 3 | 4294967267 | 3 |
control_tb       00000          00000 | 00001 | 00010 | 00011 | 00100 |
```

Here we verify the instruction XOR when control has a value of 11000. The result is calculated by taking the XOR of rs1 and rs2. The values can be verified by checking the decimal values, for rs1 = 1, rs2 = 2. The output for this operation should be 3 and this is verified by the waveform.

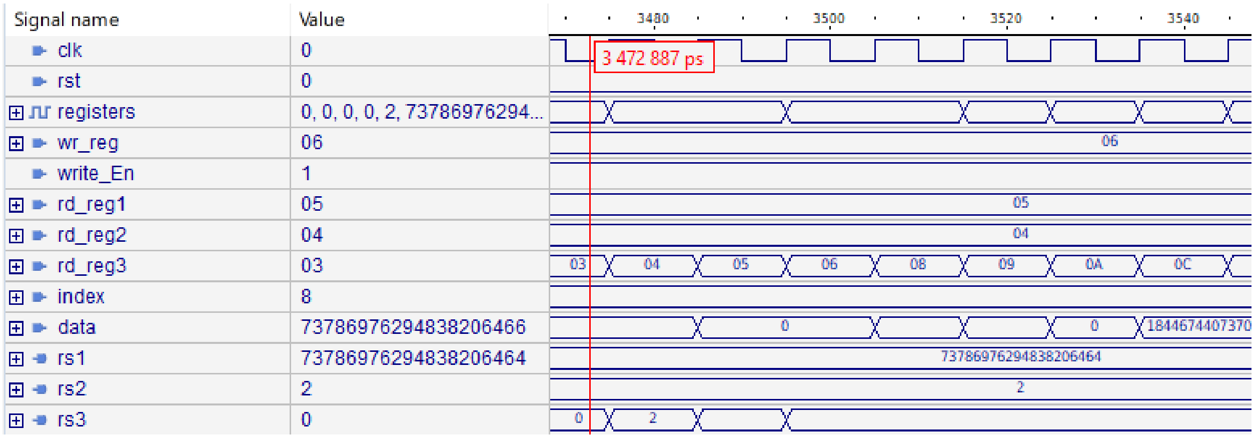


Signal name	Value	1880 1920 1960
rs1_tb	0	1
rs2_tb	0	2
rs3_tb	0	3
loadIndex_tb	011	000
immediate_tb	0003	80FC
alu_out_tb	84442493013...	3
control_tb	00000	11000

Register File

The register file has 32 128-bit registers. On any cycle, there can be 3 reads and 1 write which is denoted by signals rd_reg1, rd_reg2,rd_reg3 and wr_reg respectively. When executing instructions, each cycle two/three 128 bit- register values are read. One 128-bit result can be written if the write signal is valid. The implementation is shown below with signal write_En determining when it can be written.

```
begin

rs3 <= data when write_En = '1' and wr_reg = rd_reg3 else registers(to_integer(unsigned(rd_reg3)));
rs2 <= data when write_En = '1' and wr_reg = rd_reg2 else registers(to_integer(unsigned(rd_reg2)));
rs1 <= data when write_En = '1' and wr_reg = rd_reg1 else registers(to_integer(unsigned(rd_reg1)));

process (clk)
    begin
        if (rising_edge(clk)) then
            if write_En = '1' then
                case index is
                    when "0111" => --7
                        registers(to_integer(unsigned(wr_reg)))(127 downto 112) <= data(127 downto 112);
                    when "0110" => --6
                        registers(to_integer(unsigned(wr_reg)))(111 downto 96) <= data(111 downto 96);
                    when "0101" => --5
                        registers(to_integer(unsigned(wr_reg)))(79 downto 64) <= data(79 downto 64);
                    when "0100" => --4
                        registers(to_integer(unsigned(wr_reg)))(79 downto 64) <= data(79 downto 64);
                    when "0011" => --3
                        registers(to_integer(unsigned(wr_reg)))(47 downto 32) <= data(47 downto 32);
                    when "0010" => --2
                        registers(to_integer(unsigned(wr_reg)))(47 downto 32) <= data(47 downto 32);
                    when "0001" => --1
                        registers(to_integer(unsigned(wr_reg)))(31 downto 16) <= data(31 downto 16);
                    when "0000" => --0
                        registers(to_integer(unsigned(wr_reg)))(15 downto 0) <= data(15 downto 0);
                    when others => --full
                        registers(to_integer(unsigned(wr_reg))) <= data;
                end case;
            end if;
        end if;
```

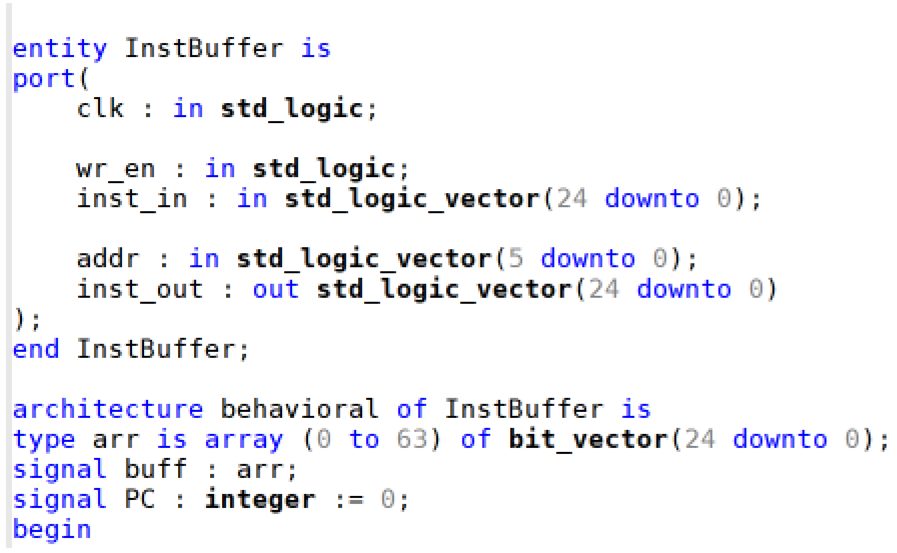


Signal name	Value
clk	0
rst	0
registers	0, 0, 0, 0, 2, 73786976294...
wr_reg	06
write_En	1
rd_reg1	05
rd_reg2	04
rd_reg3	03
index	8
data	73786976294838206466
rs1	73786976294838206464
rs2	2
rs3	0

As seen, the Register File correctly reads the register numbers inputted and outputs the values of the selected registers in the next clock cycle when write_En is 1. The values given at rd_reg1,rd_reg2, and rd_reg3 indicate the index for the array registers on which register to read. Based on that index it displays the registers corresponding to it for rs1, rs2 and rs3 respectively. Signal wr_reg here is used to determine which register is being written and corresponds to the index in the registers array. Signal data corresponds to the register number corresponding to that index.

Instruction Buffer

The instruction buffer can store 64 25-bit instructions. As seen below we have an input for the 25 bit instruction. We have an array that can store all 64 instructions into array arr.

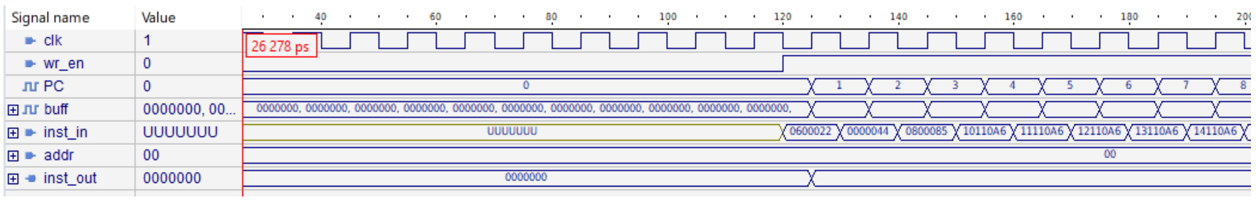


```
entity InstBuffer is
port(
    clk : in std_logic;

    wr_en : in std_logic;
    inst_in : in std_logic_vector(24 downto 0);

    addr : in std_logic_vector(5 downto 0);
    inst_out : out std_logic_vector(24 downto 0)
);
end InstBuffer;

architecture behavioral of InstBuffer is
type arr is array (0 to 63) of bit_vector(24 downto 0);
signal buff : arr;
signal PC : integer := 0;
begin
```

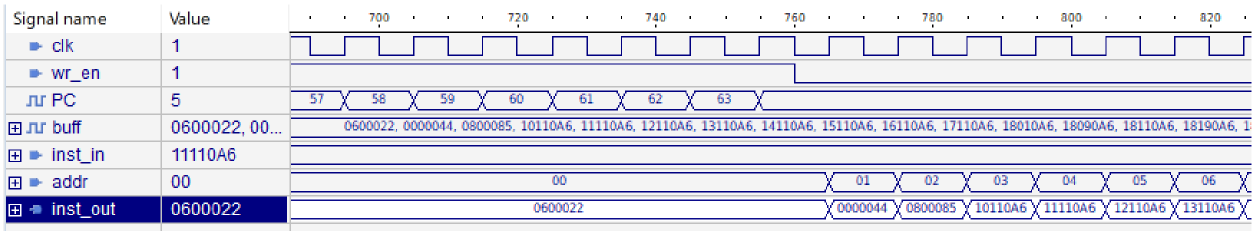
On each cycle, the instruction specified by the Program Counter (PC) is fetched, and the value of PC is incremented by 1. As seen below, the instruction is fetched on the rising edge of the clock and when write enabled is 1. After the instruction is fetched, the PC is reset back to 0 if all the instructions have been fetched otherwise it is incremented by 1.

```
begin
    process(clk)
    begin
        if rising_edge(clk) then
        if(wr_en = '1') then
            buff(PC) <= to_bitvector(inst_in);

            if(PC = 63) then
                PC <= 0;
            else
                PC <= PC + 1;
            end if;
        end if;
    end if;
    end process;

    inst_out <= to_stdlogicvector(buff(to_integer(unsigned(addr))));
end behavioral;
```

The operation of the instruction buffer can be verified below, where the instruction fetched is specified by the increment program counter. Once the program counter is reset to 0, we are able to notice at inst_out the instruction output increments with our addr signal and matches those instruction fetched by the program counter.



Signal name	Value
clk	1
wr_en	0
PC	0
buff	0000000, 00...
inst_in	UUUUUUU
addr	00
inst_out	0000000



Signal name	Value
clk	1
wr_en	1
PC	5
buff	0600022, 00...
inst_in	11110A6
addr	00
inst_out	0600022

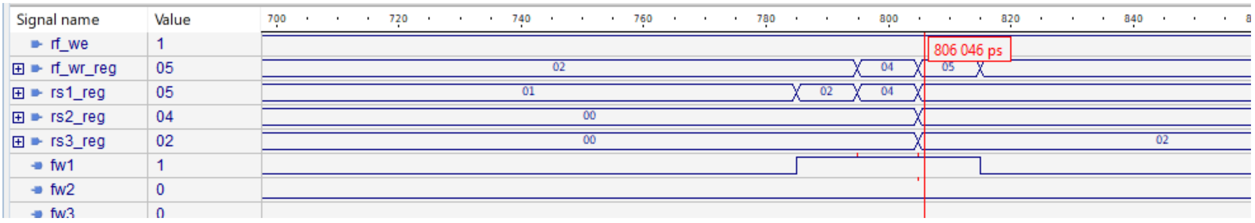
700
720
740
760
780
800
820
57
58
59
60
61
62
63
0600022, 0000044, 0800085, 10110A6, 11110A6, 12110A6, 13110A6, 14110A6, 15110A6, 16110A6, 17110A6, 18010A6, 18090A6, 18110A6, 18190A6,
00
01
02
03
04
05
06
0600022
0000044
0800085
10110A6
11110A6
12110A6
13110A6

Forwarding Unit

Every instruction uses the most recent value of a register. The forwarding unit ensures that only valid data and source/destinations registers are being considered for forwarding. As shown below, data is forwarded when signal write enable, rf_we,1 = 1 and when the most recent data to be forwarded matches the data from one of the inputs from the register. When both conditions are met, the data will be forwarded and can be represented with a value of one corresponding to which register value was forwarded.

```vhdl
architecture Behavioral of DataForwarding is
begin
process (rf_we, rf_wr_reg, rs1_reg, rs2_reg, rs3_reg)
begin
    if (rf_we = '1' and rf_wr_reg = rs1_reg) then
        fw1 <= '1';
    else
        fw1 <= '0';
    end if;

    if (rf_we = '1' and rf_wr_reg = rs2_reg) then
        fw2 <= '1';
    else
        fw2 <= '0';
    end if;

    if (rf_we = '1' and rf_wr_reg = rs3_reg) then
        fw3 <= '1';
    else
        fw3 <= '0';
    end if;
end process;
```

As shown below, we have rf_we = 1 and the most recent data having a value of 5. The value matching that comes from register 1 and results in it being forwarded which is indicated by fw1 having a value of 1 and fw2 and fw3 having a value of 0, indicating it is not forwarded.



```
Signal name   Value
rf_we         1
rf_wr_reg     05
rs1_reg       05
rs2_reg       04
rs3_reg       02
fw1           1
fw2           0
fw3           0
```

Four- Stage Pipeline

The pipeline is implemented as a structural model with modules for each corresponding pipeline stages and their interstage registers. Four instructions can be at different stages of the pipeline at every cycle and all instructions take four cycles to complete. Clock edge-sensitive pipeline registers separate the If, ID, EXE, and WB stages. Data is written to the register file after the WB stage.

Stage 1

```vhdl
--Stage 1: Instruction Fetch
ProgramCounter: reg generic map (DWIDTH => 6)
port map (
    CLK => CLK,
    RST => RESET,
    CE  => run,
    D   => adder_out,
    Q   => pc_out
);

PCadder: adder generic map (DWIDTH => 6)
port map (
    A => "000001",
    B => pc_out,
    D => adder_out
);

InstBuffer_inst: InstBuffer
port map (
    clk => CLK,

    wr_en => wr_en,
    inst_in => inst_in,

    addr => pc_out,
    inst_out => inst
);
```

Stage 2

```vhdl
--Stage 2: Decode and Read Operands

RF_inst: RF port map (
    CLK => CLK,
    rst => RESET,

    write_En => rf_we,
    index => rf_loadindex,
    data => rf_datain,
    wr_reg => rf_wr_reg,
    rd_reg1 => inst_IFID(9 downto 5),
    rd_reg2 => inst_IFID(14 downto 10),
    rd_reg3 => inst_IFID(19 downto 15),
    rs1 => rs1,
    rs2 => rs2,
    rs3 => rs3
);
```

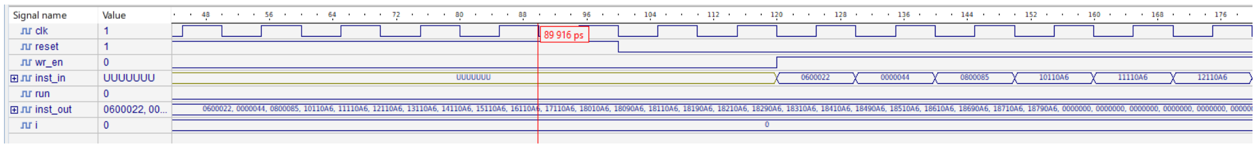
Stage 3

```vhdl
--Stage 3:Execute
ControlUnit_EX: ControlUnit
port map(
    inst => inst_IDEX,

    control => control,
    write_en => open,
    immediate => immediate,
    loadIndex => loadIndex
);
```

Stage 4

```vhdl
--stage 4: Write back
--control
ControlUnit_WB: ControlUnit
port map(
    inst => inst_EXWB,

    control => open,
    write_en => rf_we,
    immediate => open,
    loadIndex => rf_loadIndex
);
rf_datain <= alu_out_EXWB;
rf_wr_reg <= inst_EXWB(4 downto 0);

DataForwarding_inst: DataForwarding
    port map(
        rf_we => rf_we,
        rf_wr_reg => rf_wr_reg,

        rs1_reg => inst_IDEX(9 downto 5),
        rs2_reg => inst_IDEX(14 downto 10),
        rs3_reg => inst_IDEX(19 downto 15),

        fw1 => fw1,
        fw2 => fw2,
        fw3 => fw3
);
```

Pipeline Output

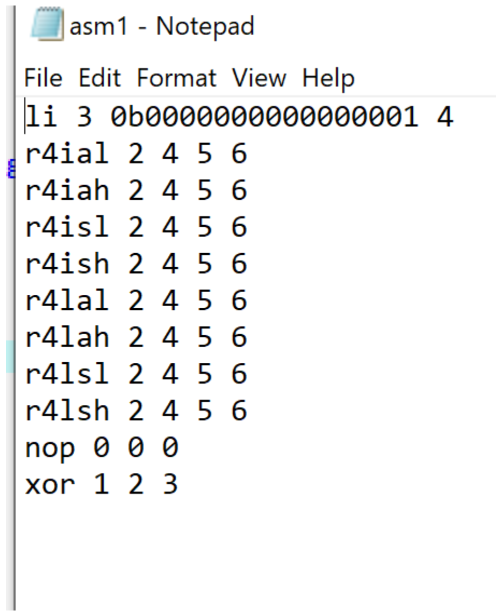


```
Signal name   Value
clk           1
reset         1
wr_en         0
inst_in       UUUUUUU
run           0
inst_out      0600022, 00...
i             0
```

48
56
64
72
80
88
96
104
112
120
128
136
144
152
160
168
176
89 916 ps
UUUUUUU
0600022
0000044
0800085
10110A6
11110A6
12110A6
0600022, 0000044, 0800085, 10110A6, 11110A6, 12110A6, 13110A6, 14110A6, 15110A6, 16110A6, 17110A6, 18010A6, 18090A6, 18110A6, 18190A6, 18210A6, 18290A6, 18310A6, 18410A6, 18490A6, 18510A6, 18610A6, 18690A6, 18710A6, 18790A6, 0000000, 0000000, 0000000, 0000000, 0000000, 000000
0

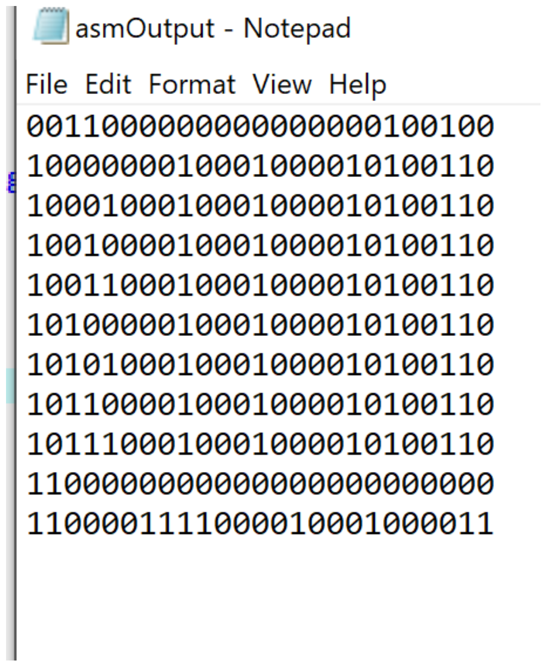
Assembler

The assembler was a separate program. It converts an assembly file to the binary format for the instruction buffer.



```
li 3 0b0000000000000001 4
r4ial 2 4 5 6
r4iah 2 4 5 6
r4isl 2 4 5 6
r4ish 2 4 5 6
r4lal 2 4 5 6
r4lah 2 4 5 6
r4lsl 2 4 5 6
r4lsh 2 4 5 6
nop 0 0 0
xor 1 2 3
```

Output for Assembler



```
0011000000000000000100100
1000000010001000010100110
1000100010001000010100110
1001000010001000010100110
1001100010001000010100110
1010000010001000010100110
1010100010001000010100110
1011000010001000010100110
1011100010001000010100110
1100000000000000000000000
1100001111000010001000011
```

Conclusion:

The finished product of the pipeline was successful in properly pushing data through and handling multiple instructions at once. The individual units and registers in the pipeline are properly functioning as shown above in the simulations. The pipeline contained core units such as the Instruction Buffer, Register File, and the ALU. Interstate registers carried data to each unit to be processed, where it was then passed onto the next register. The goal of the pipeline was to allow all major units to be busy at all times by acting as an assembly line. Multiple instructions are able to be processed by splitting the workload and simply moving the data through the pipeline. The block diagram illustrates how the pipeline should act with data transfer. The only part I wasn't able to accomplish was to have a results file that highlights all the important signals for all stages of the pipeline, for all cycles of execution.