Pipelined Multimedia Unit Design with VHDL Hardware Description Language

ESE 345
Fall 2017
Professor Mikhail Dorojevets

Prepared By:
Reynerio Rubio (109899097)

John Kim (109622176)

Introduction

Purpose

The purpose of this project was to use VHDL/Verilog and modern CAD tools to create hierarchical gate-level and dataflow/RTL design for a three-stage pipelined multimedia Multimedia Unit. This Multimedia unit has a reduced set of instructions, which is similar to those found in the Sony Cell SPU architecture.

Project Description:

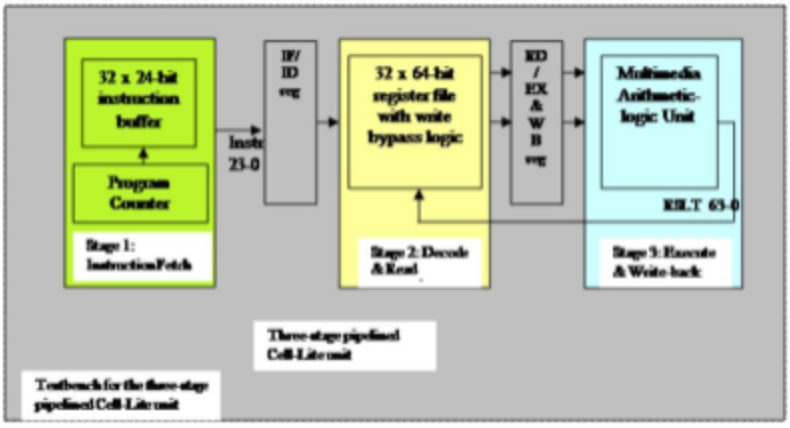

Figure 1.1 - Three-stage Pipelined Cell-Lite Unit

The Multimedia unit uses a three-stage pipeline, as shown in ***Figure 1.1***, where each stages completes a section of each executed instruction allowing the system to improve efficiency. The initial stage communicates with the program counter to fetch instructions and passes it along to the second stage. The second stage then decodes the fetched instruction into the proper instruction format and from there the decoded instruction gets passed into the third stage.

The third stage then executes the instruction according to the opcode and pushes the result through to the writeback, which is then written to the rd register. In between each stage are buffers, each holds the memory for the program in the case a potential data hazard is discovered. The buffers allow for data forwarding in order to prevent the data hazard and to allow the pipeline to function as intended.

Instruction Format:

Every instruction is 24 bits and there are 3 types of instructions. The first type is the I-type instruction, as shown by ***Figure 2.1***. The 23rd bit is a 1, signifying it is of I-type, then bit 22 to 21 signifies the byte of rd the 16-bit immediate value will be placed into. Bits 20 to 5 are the 16-bit immediate value that will be loaded into rd and then bits 4 to 0 specify the rd register.

The R3-type instruction, shown by ***Figure 2.2***, can be detected by looking at bits 23 to 22 which have to be "00". Bits 21 to 15 signify the opcode which tells the Multimedia Unit the actual instruction to be executed. Bits 14 to 10, 9 to 5, and 4 to 0 represent the rs2, rs1, and rd registers respectively. All R3-type instructions can be seen in ***Figure 6.1***.

The R4-type instruction, shown by *Figure 2.3*, can be detected by looking at bits 23 to 22, which have to be "01". Bits 21 to 20 represent the MA/MS/I/h, which signifies which instruction to execute. Bits 19 to 15, 14 to 10, 9 to 5, and 4 to 0 represent the rs3, rs2, rs1, and rd registers respectively. All R4-type instructions can be seen in ***Figure 6.2***.

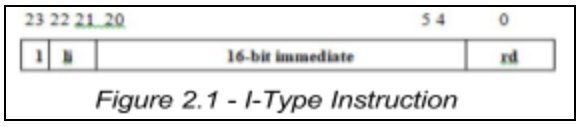

Figure 2.1 - I-Type Instruction

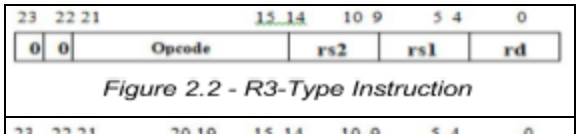

Figure 2.2 - R3-Type Instruction

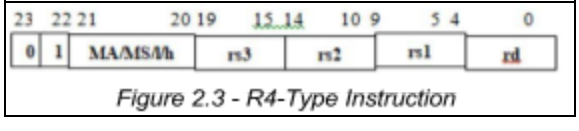

Figure 2.3 - R4-Type Instruction

Implementation

Design Implementation:

The Multimedia unit was implemented in VHDL (Very-high speed Hardware Description Language) using Active-HDL Student Edition. The Register file, pipeline, and buffers were implemented as a behavioral model and each stage within the pipeline was implemented to be sensitive to the rising and falling edges of the clock. ***Figure 3.0*** below shows our implementation for the three stage pipeline.

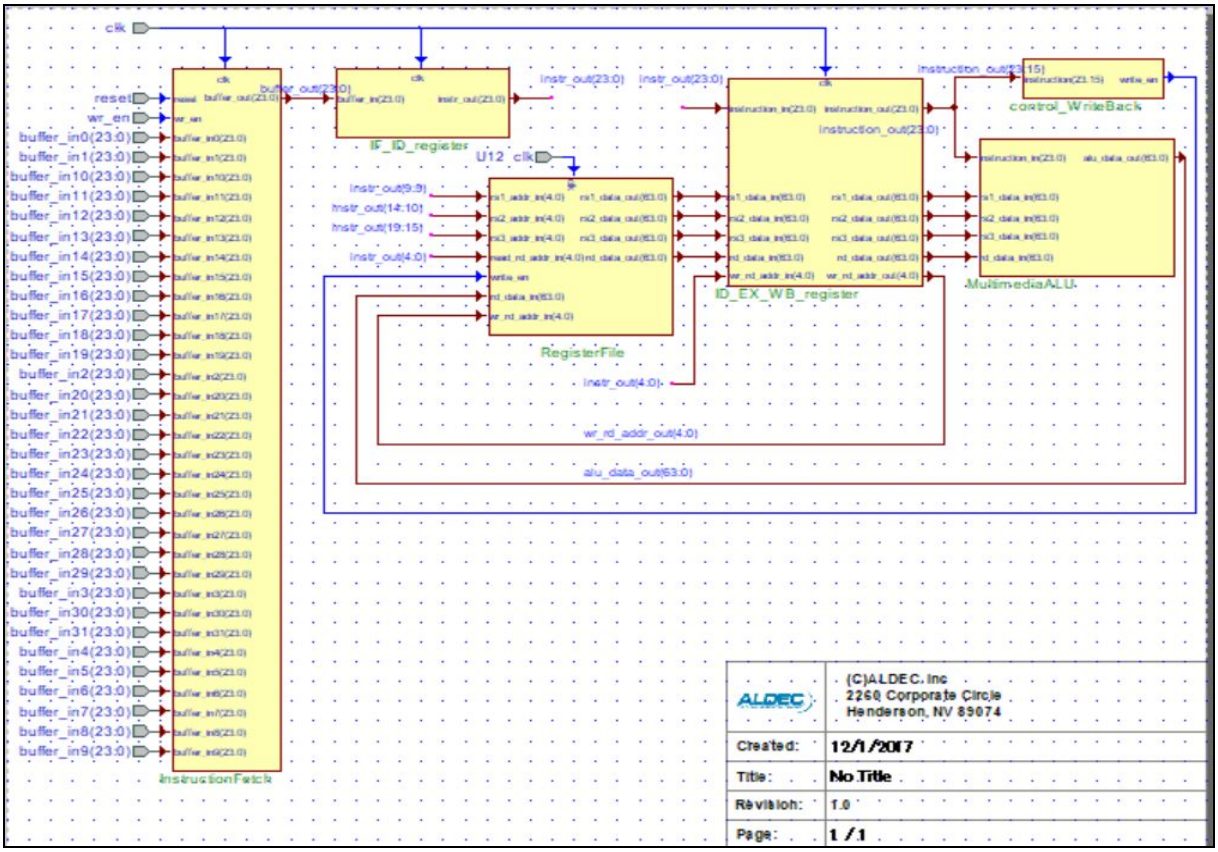

Figure 3.0 : Multimedia Pipelined Unit

The Multimedia ALU was implemented using a structural/gate level design. *Figure 3.1* below shows the Multimedia ALU Block Diagram. The Multimedia takes in 5 inputs: instruction_in(23:0), rs1_data_in(63:0), rs2_data_in(63:0), rs3_data_in(63:0), and rd_data_in(63:0). The Multimedia ALU outputs 1 value: alu_data_out(63:0). The Multimedia ALU was implemented with several more components, which can be seen in *Figure 3.2.*

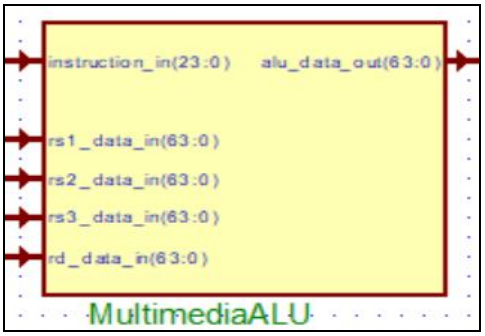

Figure 3.1 : Multimedia ALU Unit

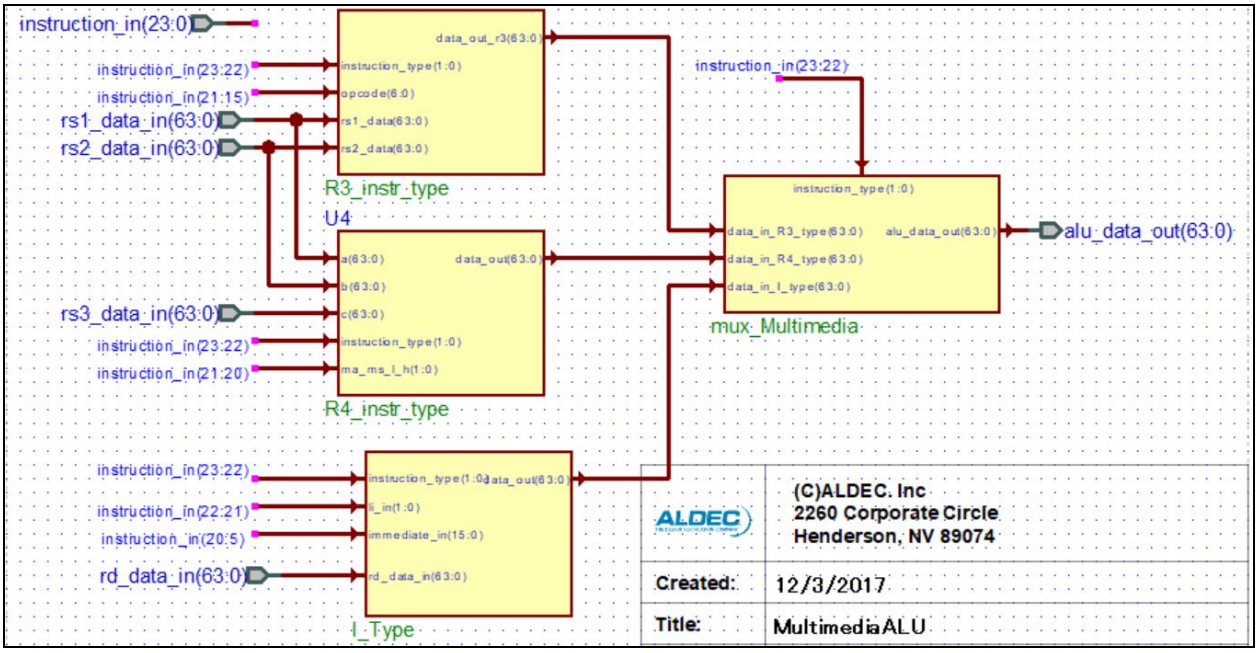

Figure 3.2 : Multimedia ALU Unit Implementation

To implement our Multimedia ALU Unit, we created three separate logic blocks for each of the three instruction formats. These instructions are R3-type, R4-type, and I-type. For each instruction, its type is defined by the bits instruction_in(23:22). The R3 instruction type unit takes in the following inputs: instruction_type, instruction_in(21:5) for the opcode, rs1_data_in(63:0), and rs2_data_in(63:0). The R4 instruction type unit takes in the following inputs: instruction_type, instruction_in(21:20) for the MA/MS/l/h bits, rs1_data_in(63:0), rs2_data_in(63:0), and rs3_data_in(63:0). The R1/immediate instruction type unit takes in the following inputs: : instruction_type, instruction_in(22:21) for the "li" bits, instruction_in(20:5) for the 16-bit immediate value, and rd_data_in(63:0).

The instructions for the R3 instruction type were split up into two different units: the LSRUnit (LogicalShiftRotate) unit and the Arithmetic Unit. *Figure 3.3* below shows our implementation for the R3 instruction type. The LSRUnit implements the following instructions: nop (no operation), bcw (broadcast word), and (logical bitwise and), or (logical or), popcnth (count ones in halfword), clz (count leading zeros), rot (rotate right), shlhi (shift left halfword immediate). These instructions are similar in that their opcode bit, opcode(3) = '0'. The ArithmeticUnit implements the following instructions: a (add word), sfw (subtract from word), ah (add halfword), sfh (subtract from halfword), ahs (add halfword saturated), sfhs (subtract from halfword saturated). These instructions are similar in that their opcode bit, opcode(3) = '1'. The multiplexer following these units selects data depending on the opcode bit, opcode(3). Here, the multiplexer selects the data from the LSRUnit if opcode bit, opcode(3) = '0' or data from the ArithmeticUnit if opcode bit, opcode(3) = '1'.

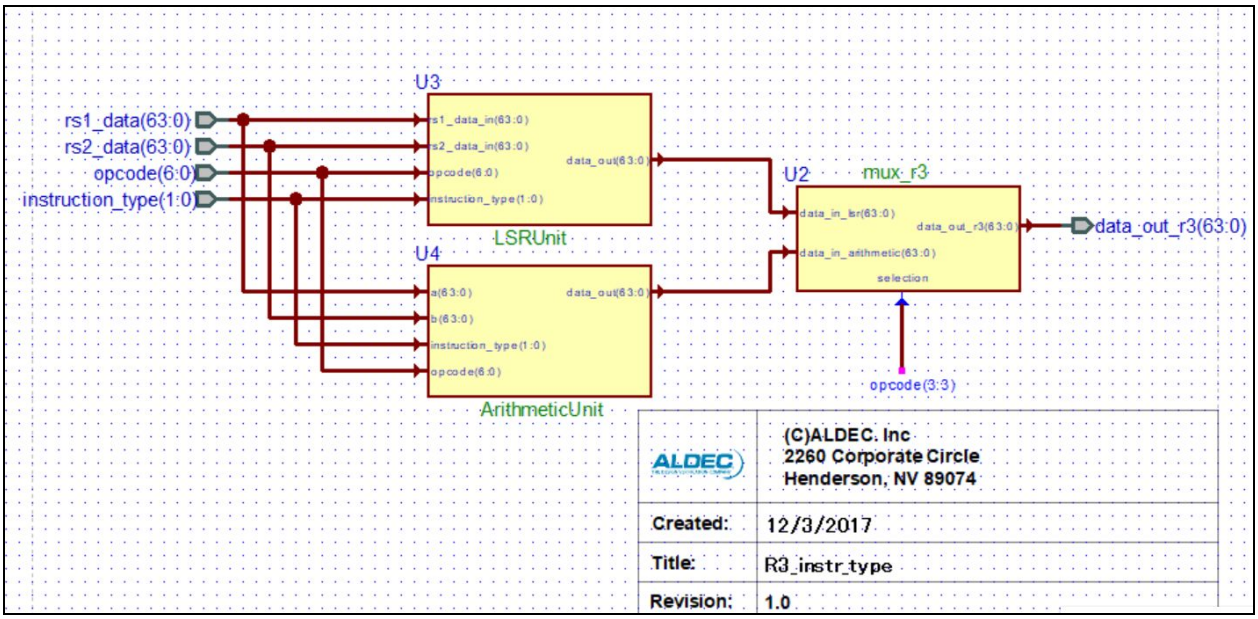

Figure 3.3 - R3-Type Unit Implementation

Instruction Buffer

The instruction buffer receives 32 24-bit instructions. The instruction buffer is then connected to a program counter, which is connected to the testbench, so that every cycle an instruction will be taken in and the program counter will be incremented to point to the next instruction.

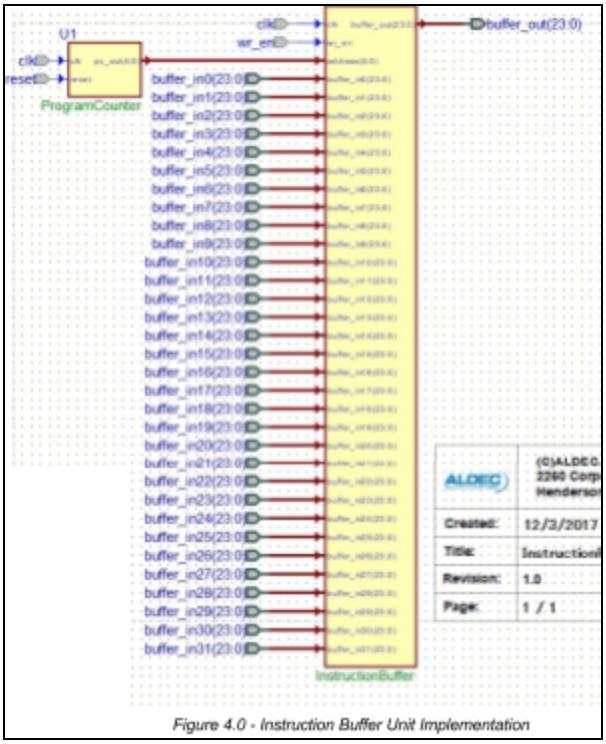

Figure 4.0 - Instruction Buffer Unit Implementation

Register File

The Register File contains 32 64-bit registers and at every rising edge of the clock, register values are read. It is responsible for reading the data stored in each register as well as writing to the rd register, assuming that the write-back signal is enabled. ***Figure 4.1*** below shows our implementation of the Register File.

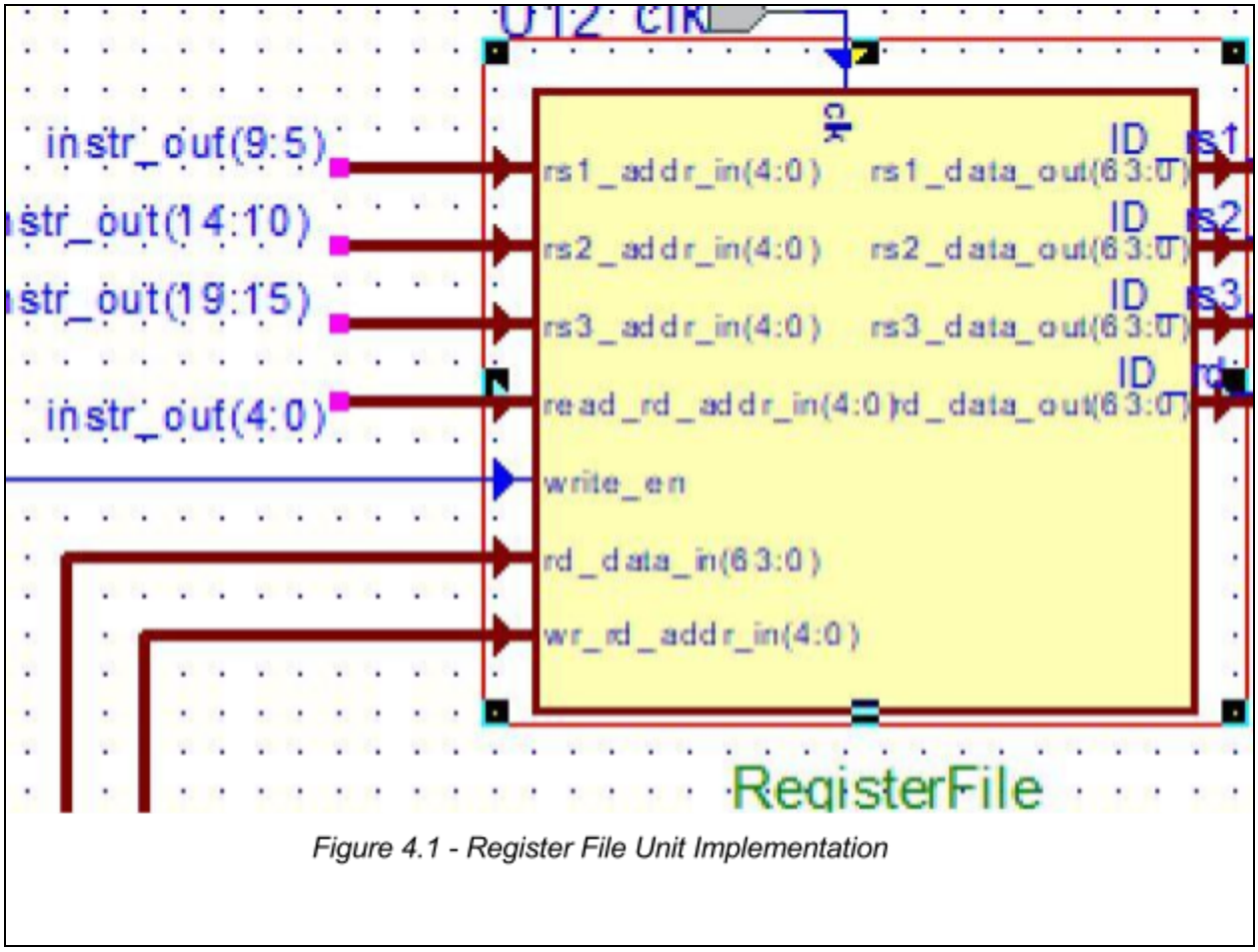

Figure 4.1 - Register File Unit Implementation

Forwarding Unit

The forwarding unit is designed to provide protection against data hazards and is used to allow the three-stage pipeline to function with a minimal number of stalls. The unit is used to pass the ALU output to the input of a multiplexor. The other input for the multiplexor is

connected to the output of the register file and from there a select channel decides whether the forwarded ALU output is needed or not and chooses accordingly. This allows the pipeline to use data that has been changed, without having to wait another cycle for the writeback to occur. This ensures that the minimal number of stalls are used and that each instruction is executed in three clock cycles. The implementation for the forwarding unit is shown in the code snippet from the RegisterUnit entity below in ***Figure 4.2***

```
begin
    Process (clk, rd_data_in, rs1_addr_in, rs2_addr_in, rs3_addr_in)
    begin
        -- USE ADDRESS TO GET DATA FROM REGISTER FILE
        rs1_data_sig <= RegisterFile_sig(to_integer(unsigned(rs1_addr_in)));
        rs2_data_sig <= RegisterFile_sig(to_integer(unsigned(rs2_addr_in)));
        rs3_data_sig <= RegisterFile_sig(to_integer(unsigned(rs3_addr_in)));
        rd_data_sig  <= RegisterFile_sig(to_integer(unsigned(read_rd_addr_in)));

        -- WRITE TO RD
        if (falling_edge(clk) and write_en ='1' ) then
            RegisterFile_sig(to_integer(unsigned(wr_rd_addr_in))) <= rd_data_in;
        end if;

        -- SELECT WRITE BYPASS FOR RS1 ('1' == write bypass, '0' == normal)
        if rs1_addr_in = wr_rd_addr_in then
            select_rs1 <= '1';
        else
            select_rs1 <= '0';
        end if ;
        -- SELECT WRITE BYPASS FOR RS2 ('1' == write bypass, '0' == normal)
        if rs2_addr_in = wr_rd_addr_in then
            select_rs2 <= '1';

        else
            select_rs2 <= '0';
        end if
        -- SELECT WRITE BYPASS FOR RS3 ('1' == write bypass, '0' == normal)
        if rs3_addr_in = wr_rd_addr_in then
            select_rs3 <= '1';
        else
            select_rs3 <= '0';
        end if ;
        -- SELECT WRITE BYPASS FOR RD ('1' == write bypass, '0' == normal)
        if read_rd_addr_in = wr_rd_addr_in then
            select_rd <= '1';
        else
            select_rd <= '0';
        end if ;

    end process;
    -- PORT MAPPING FOR RS1, RS2, RS3
    mux_rs1: entity mux_RegisterFile port map (normal => rs1_data_sig, write_bypass=> rd_data_in, selection => select_rs1, data_out => rs1_data_out);
    mux_rs2: entity mux_RegisterFile port map (normal => rs2_data_sig, write_bypass=> rd_data_in, selection => select_rs2, data_out => rs2_data_out);
    mux_rs3: entity mux_RegisterFile port map (normal => rs3_data_sig, write_bypass=> rd_data_in, selection => select_rs3, data_out => rs3_data_out);
    mux_rd : entity mux_RegisterFile port map (normal => rd_data_sig, write_bypass=> rd_data_in, selection => select_rd, data_out => rd_data_out);

end behavioral;
```

Figure 4.2 - Data Forwarding and Write Back Logic

Pipeline

The Multimedia Unit contains two pipelines, the IF/ID Reg and the ID/EX Reg. The program counter fetches and outputs all of the instructions into the IF/ID Reg. The IF/ID Reg then passes the necessary bits into the Register File so that it can handle the registers accordingly. The Register File then gets the respective 64-bit data for each register and passes it along with the opcode to the ID/EX Reg. The IF/ID Reg then stores the data and outputs all of the data into the Multimedia ALU, which is used for all computations based on the instruction type and opcode. For example, the Arithmetic module is used for all arithmetic instructions and the LSR module is used for all logical shift and rotate instructions. There are additional modules used, the I_Type, R3_Type, and the R4_Type, which have select channels connected to bits 23 to 22. Each module is selected when the corresponding bits match the conditional statements and from there the data and corresponding registers get pushed into the rest of the ALU to execute the instruction. Since all the instructions, except nop, store their values into rd, writeback is enabled and the ALU output is written back to the register file, to be written into register rd.

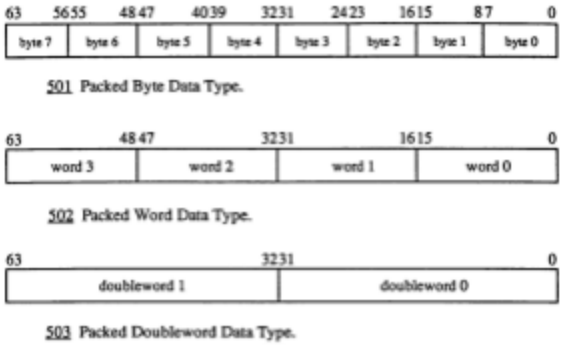

Figure 5.0 - Data Types

Instructions:

The instructions used in the Multimedia Unit are of three types, I-type, R3-type, and R4-type. The format for each figure can be seen in *Figure 2.1 - 2.3* respectively. The descriptions for each R3-type and R4-type instruction can be seen in *Figure 6.1* and *Figure 6.2* respectively. There are three stages to the Multimedia Unit so each instruction should take three cycles to execute.

Opcode 21-15	Description of Instruction Opcode
xxx0000	**Nop**
xxx0001	**bcw**: broadcast a right 32-bit word of register **rs1** to both left and right 32-bit words of register **rd.**
xxx0010	**and**: bitwise ***logical and*** of the contents of registers ***rs1*** and ***rs2***
xxx0011	**or**: bitwise ***logical or*** of the contents of registers ***rs1*** and ***rs2***
xxx0100	**popcnth**: *count ones in halfwords*: the number of 1s in each of the four halfword-slots in register ***rs1*** is computed. If the halfword slot in register ***rs1*** is zero, the result is 0. Each of the results is placed into corresponding 16-bit slot in register ***rd***. ***(Comments: 4 separate 16-bit halfword values in each 64-bit register)***
xxx0101	**clz**: *count leading zeroes in words*: for each of the two 32-bit word slots in register ***rs1*** the number of zero bits to the left of the first non-zero bit is computed. If the word slot in register ***rs1*** is zero, the result is 32. The two results are placed into the corresponding 32-bit word slots in register ***rd***. ***(Comments: 2 separate 32-bit values in each 64-bit register)***
xxx0110	**rot:** *rotate right*: the contents of register ***rs1*** are rotated to the right according to the count in the 6 least significant bits (5 to 0) of the contents of register ***rs2***. The result is placed in register ***rd***. If the count is zero, the contents of register ***rs1*** are copied unchanged into register ***rd***. Bits rotated out of the right end of the 64-bit contents of register ***rs1*** are rotated in at the left end.
xxx0111	**shlhi**: *shift left halfword immediate*: packed 16-bit halfword shift left logical of the contents of register ***rs1*** by the 4-bit immediate value of instruction field **rs2**. Each of the results is placed into the corresponding 16-bit slot in register ***rd***. ***(Comments: 4 separate 16-bit values in each 64-bit register)***
xxx1000	**a**: *add word*: packed 32-bit unsigned add of the contents of registers ***rs1*** and ***rs2*** ***(Comments: 2 separate 32-bit values in each 64-bit register)***
xxx1001	**sfw**: *subtract from word*: (packed) 32-bit unsigned subtract of the contents of registers ***rs1*** and ***rs2*** ***(Comments: 2 separate 32-bit values in each 64-bit register)***
xxx1010	**ah**: *add halfword* : (packed) (16-bit) halfword unsigned add of the contents of registers ***rs1*** and ***rs2*** ***(Comments: 4 separate 16-bit values in each 64-bit register)***
xxx1011	**sfh**: *subtract from halfword*: (packed) (16-bit) halfword unsigned subtract of the contents of registers ***rs1*** and ***rs2***
xxx1100	**ahs**: *add halfword saturated*: (packed) (16-bit) halfword signed add **with saturation** of the contents of registers ***rs1*** and ***rs2***
xxx1101	**sfhs**: *subtract from halfword saturated*: (packed) (16-bit) signed subtract **with saturation** of the contents of registers ***rs1*** and ***rs2***
xxx1110	**mpyu**: *multiply unsigned*: the 16 rightmost bits of each of the two 32-bit slots in registers ***rs1*** are multiplied by the 16 rightmost bits of the corresponding 32-bit slots

MA/MS/l/h 21-20	Description of instruction code [21-20]
00	**Signed integer multiple-add low with saturation:** Multiply low 16-bit-fields of each 32-bit field of registers **rs3** and **rs2,** then add 32-bit products to 32-bit fields of register **rs1,** and save result in register **rd**.
01	**Signed integer multiple-add high with saturation:** Multiply high 16-bit-fields of each 32-bit field of registers **rs3** and **rs2,** then add 32-bit products to 32-bit fields of register **rs1,** and save result in register **rd**.
10	**Signed integer multiple-subtract low with saturation:** Multiply low 16-bit-fields of each 32-bit field of registers **rs3** and **rs2,** then subtract 32-bit products from 32-bit fields of register **rs1,** and save result in register **rd**.
11	**Signed integer multiple-subtract high with saturation:** Multiply high 16-bit-fields of each 32-bit field of registers **rs3** and **rs2,** then subtract 32-bit products from 32-bit fields of register **rs1,** and save result in register **rd**.

Figure 6.2 - R4-Type Instructions

Simulation Results

The simulations run can be seen on ***Figure 7.0***, which shows the Opcode pushed into the instruction buffer (in order), the instruction it executes, and the updated rd value. As you can see, every given instruction executes and outputs the expected value. On top of that, the pipeline executes every single instruction in three clock cycles so there is never any delay between one writeback to the next nor any data hazard issues. Moreover, ***Figures 7.1 - 7.4*** show several waveforms of the entire system while executing these instructions.

Opcode (Hex)	Instruction	$15 Value	$12 Value	$1 Value	$13 Value	$2 Value	$6 Value
BFFFCF	li 1($15), FFFE	0000 0000 FFFE 0000					
0081EC	bcw $12, $15	0000 0000 FFFE 0000	FFFE 0000 FFFE 0000				
04318C	a $12, $12, $12	0000 0000 FFFE 0000	FFFC 0000 FFFC 0000				
DFFFEF	li 2($15), FFFF	0000 FFFF FFFE 0000	FFFC 0000 FFFC 0000				
04B1E1	sfw $1, $15, $12	0000 FFFF FFFE 0000	FFFC 0000 FFFC 0000	0000 0000 0002 0000			
0131E1.	and $1, $15, $12	0000 FFFF FFFE 0000	FFFC 0000 FFFC 0000	0000 0000 FFFC 0000			
01B1E1	or $1, $15, $12	0000 FFFF FFFE 0000	FFFC 0000 FFFC 0000	FFFC FFFF FFFE 0000			
0201E1.	popcnth $1, $15	0000 FFFF FFFE 0000	FFFC 0000 FFFC 0000	0000 0010 000F 0000			
0281E1.	clz $1, $15	0000 FFFF FFFE 0000	FFFC 0000 FFFC 0000	0000 0010 0000 0000			
80002D	li 0($13), 0001	0000 FFFF FFFE 0000	FFFC 0000 FFFC 0000	0000 0010 0000 0000	0000 0000 0000 0001		
0335E1.	rot $1, $15, $13	0000 FFFF FFFE 0000	FFFC 0000 FFFC 0000	0000 7FFF FFFF 0000	0000 0000 0000 0001		
0385E1.	shlhi $1, $15, $13	0000 FFFF FFFE 0000	FFFC 0000 FFFC 0000	0000 FFFE FFFC 0000	0000 0000 0000 0001		
053C2C	ah $12, $15, $1	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE FFFC 0000	0000 0000 0000 0001		
05BD8D	sfh $13, $12, $15	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE FFFC 0000	0000 0000 0000 0000		
900001 -	li 0($1), 8000	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE FFFC 8000	0000 0000 0000 0000		
AFFFC1	li 1($1), 7FFE	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 0000 0000 0000		
A001ED	li 1($13), 000F	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 0000 000F 0000		
06358D	ahs $13, $12, $13	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 FFFD 0009 0000		
06B1ED	sfhs $13, $15, $12	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 0002 0004 0000		
955542 -	li 0($2), AAAA	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 0002 0004 0000	0000 0000 AAAA 0000	
CAAAA2	li 2($2), 5555	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 0002 0004 0000	5555 0000 AAAA 0000	
070842 -	mpyu $2, $2, $2	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 0002 0004 0000	1C71 8E39 71C8 38E4	
0775ED	absdb $13, $15, $1	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 0001 8000 8000	1C71 8E39 71C8 38E4	
07BC2D	absdb $13, $1, $15	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 0001 8000 8000	1C71 8E39 71C8 38E4	
A4000E	li 1($14), 2000	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 0001 8000 8000	1C71 8E39 71C8 38E4	
5739C2	MAh $2, $15, $15, $15	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 0001 8000 8000	0000 0000 2400 0000	
5738C2	MAh $2, $15, $15, $0	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 0001 8000 8000	0000 0000 0400 0000	
7738C0	MSh $2, $15, $15, $0	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 0001 8000 8000	0000 0000 FC00 0000	
C000A8	li 2($6), 0005	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 0001 8000 8000	0000 0000 FC00 0000	0000 0005 0000 0000
8000C6	li 0($6), 0006	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 0001 8000 8000	0000 0000 FC00 0000	0000 0005 0000 0006
4318C2	MAI $2, $6, $6, $6	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 0001 8000 8000	0000 0023 0000 002A	0000 0005 0000 0006
6318C2	MSI $2, $6, $6, $6	0000 FFFF FFFE 0000	0000 FFFD FFFA 0000	0000 FFFE 7FFE 8000	0000 0001 8000 8000	FFFF FFEC FFFF FFE2	0000 0005 0000 0006

Figure 7.0 - Simulation Results

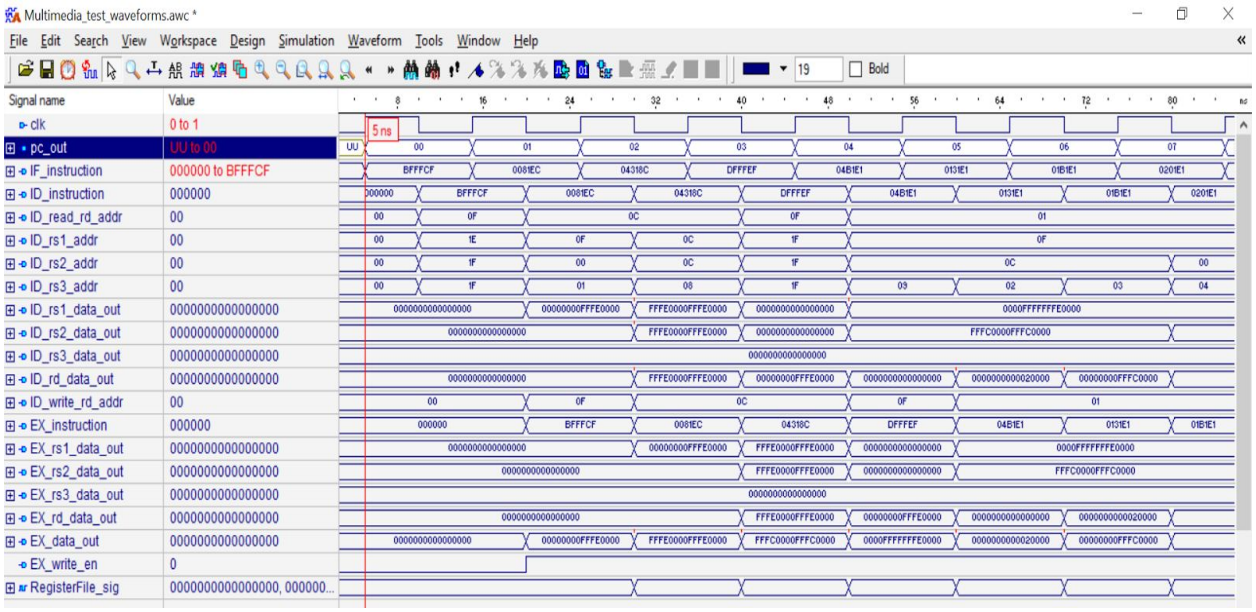

Figure 7.1 : Simulation Results (PC Count 0 through 7)

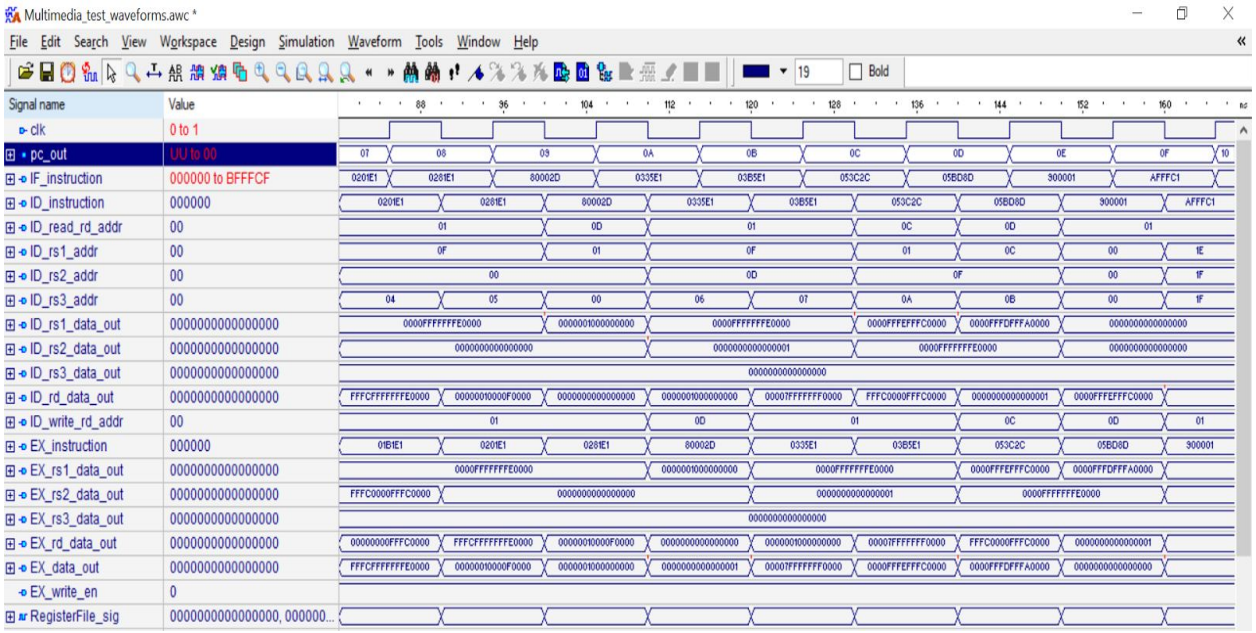

Figure 7.2 : Simulation Results (PC Count 8 through 15)

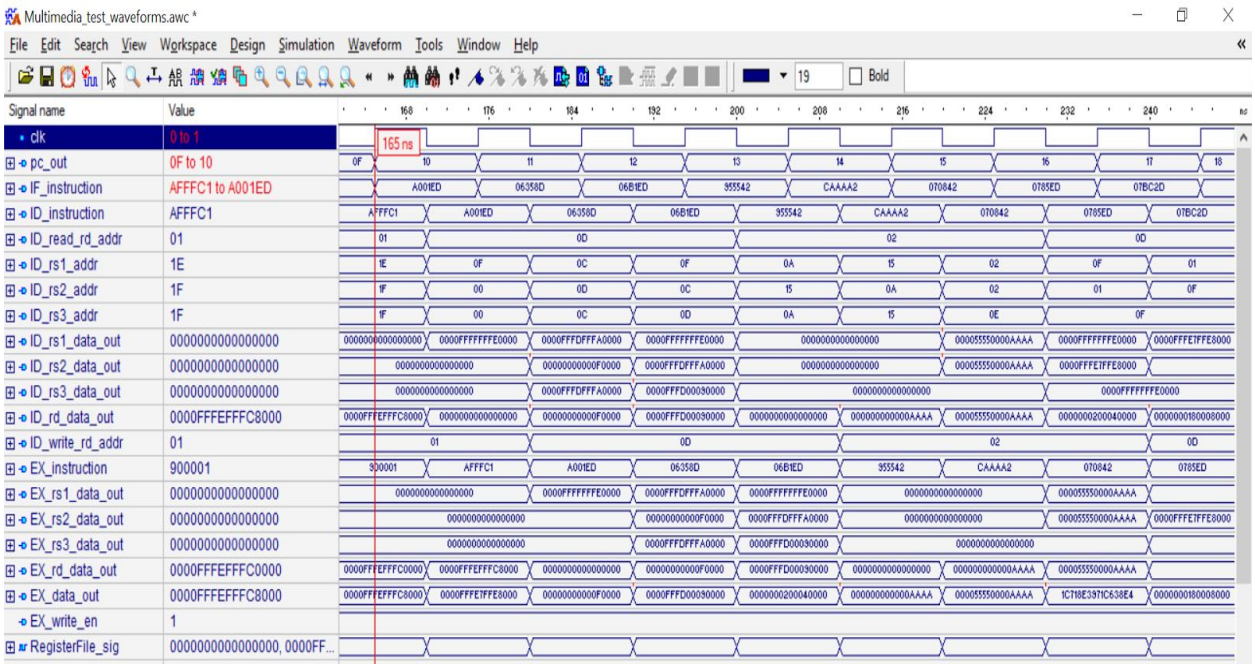

Figure 7.3 : Simulation Results (PC Count 16 through 23)

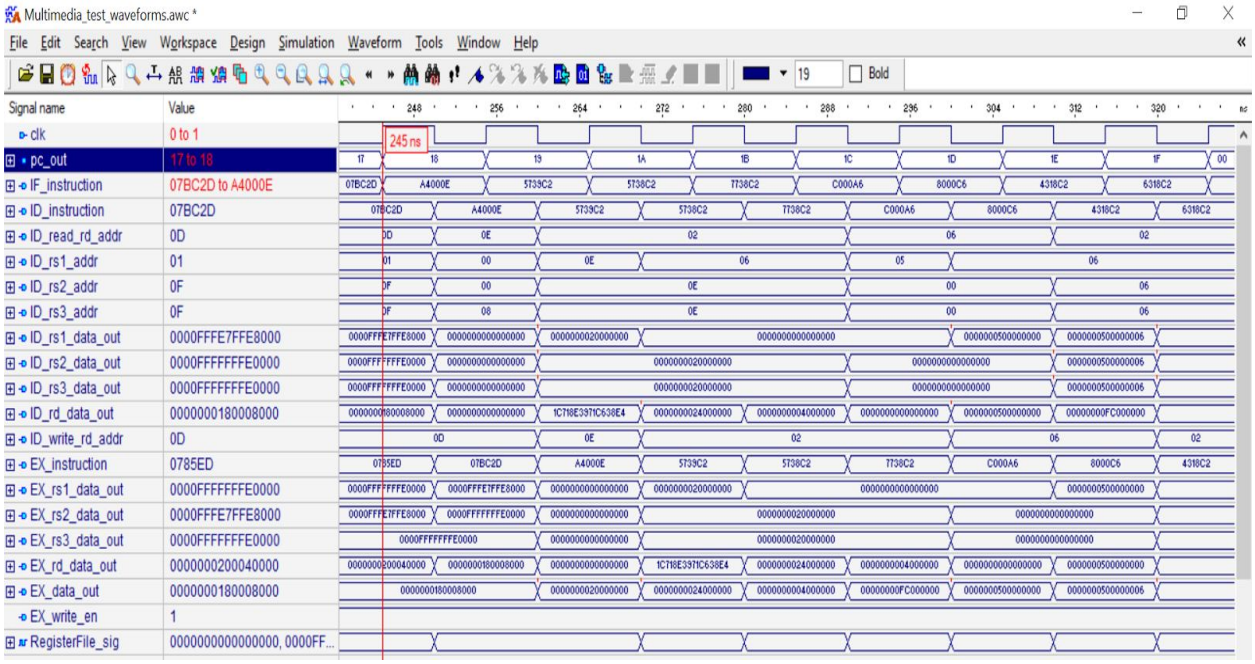

Figure 7.4 : Simulation Results (PC Count 24 through 31)

The final values stored inside the Register File is shown below in ***Figure 7.5.*** The figure was obtained using the waveforms at the end of simulation. After using 32 instructions to test the operations, we used registers 1,2, 6, 12, 13, 14, and 15. Comparing these simulated values, with the calculated expected final values of the registers previously shown in ***Figure 7.1*** we see that these values match Here, we have that register 1 = 0000 FFFE 7FFE 8000, register 2 = FFFF FFEC FFFF FFE2, register 6 = 0000 0005 0000 0006, register 12 = 0000 FFFD FFFA 0000, register 13 = 0000 0001 8000 8000, register 14 = 0000 0000 2000 0000, and register 15 = 0000 FFFF FFFE 0000.

Signal name	Value
EX_data_out	0000FFFFFFFE0000
EX_write_en	1
RegisterFile_sig	0000000000000000, 0000FF...
RegisterFile_sig[0]	0000000000000000
RegisterFile_sig[1]	0000FFFE7FFE8000
RegisterFile_sig[2]	FFFFFFECFFFFFFE2
RegisterFile_sig[3]	0000000000000000
RegisterFile_sig[4]	0000000000000000
RegisterFile_sig[5]	0000000000000000
RegisterFile_sig[6]	0000000500000006
RegisterFile_sig[7]	0000000000000000
RegisterFile_sig[8]	0000000000000000
RegisterFile_sig[9]	0000000000000000
RegisterFile_sig[10]	0000000000000000
RegisterFile_sig[11]	0000000000000000
RegisterFile_sig[12]	0000FFFDFFFA0000
RegisterFile_sig[13]	0000000180008000
RegisterFile_sig[14]	0000000020000000
RegisterFile_sig[15]	0000FFFFFFFE0000
RegisterFile_sig[16]	0000000000000000
RegisterFile_sig[17]	0000000000000000
RegisterFile_sig[18]	0000000000000000
RegisterFile_sig[19]	0000000000000000
RegisterFile_sig[20]	0000000000000000
RegisterFile_sig[21]	0000000000000000
RegisterFile_sig[22]	0000000000000000
RegisterFile_sig[23]	0000000000000000
RegisterFile_sig[24]	0000000000000000
RegisterFile_sig[25]	0000000000000000
RegisterFile_sig[26]	0000000000000000
RegisterFile_sig[27]	0000000000000000
RegisterFile_sig[28]	0000000000000000
RegisterFile_sig[29]	0000000000000000
RegisterFile_sig[30]	0000000000000000
RegisterFile_sig[31]	0000000000000000

Figure 7.5 : Register File Data

Figure 7.6 below shows how the additions to the Multimedia Unit Diagram when outputting the values from the Instruction Fetch Stage, Instruction Decode Stage, and Execution/Write Back stage. Compared to the original Multimedia Unit Diagram previously shown in ***Figure 3.0,*** the figure below only has extra output bus terminals. These outputs were used to generate the "results.txt". This file contains the status of the pipeline with the instruction opcodes, input operands, and results of execution of instructions in the pipeline for each cycle. This .txt file is included with the report.

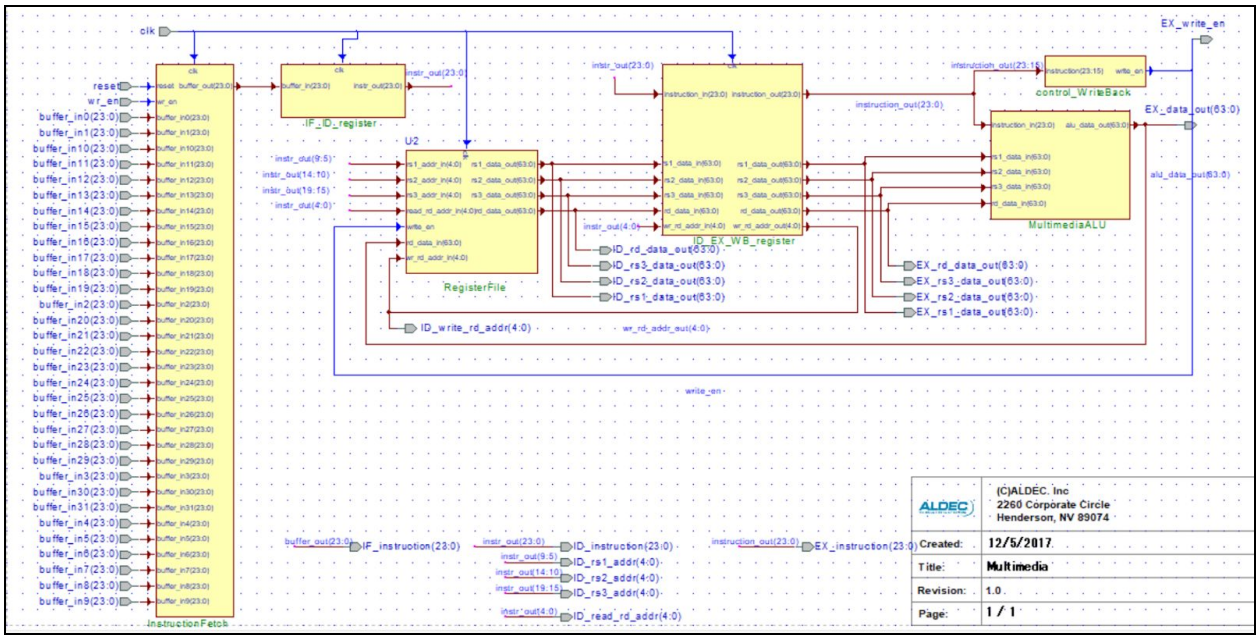

Figure 7.6 : Modified Multimedia Pipelined Unit

Conclusion

The implementation of the Multimedia Unit functioned as intended. Since the Multimedia Unit is a three-stage pipelined design, each instruction should and does execute in three clock cycles. The forwarding units eliminates any data hazards while the forwarding units

reduce/eliminate the use of stalls needed to execute certain instructions in succession. All instructions shown in *Figure 6.1* and *Figure 6.2* as well as the li instruction were implemented and executed properly. The Multimedia unit functioned as intended and executes in the fashion specified in the project description.