Pipelined SIMD Multimedia Unit Design with VHDL

ESE 345 Spring 2020
Wesley Vo (111643400)
Zachary Wong (111737587)

Objective

The goal of this project was to learn to use a hardware description language to construct a structural and behavioral design of a four-stage pipelined multimedia unit (PMMU). This multimedia unit was designed with a reduced set of multimedia instructions similar to those in the Sony Cell SPU and Intel SSE architectures. The hardware description language used was VHDL.

Context

Figure 1 was provided as a rough reference to how the four-stage PMMU should function. Each stage takes one clock cycle to execute a specific part of an instruction as instructions are fed into the system. Stage 1 is where the system fetches the instruction from the Program Counter, or PC, and increments the PC to get ready to fetch the next instruction on the next clock cycle. Stage 2 is where the passed instruction is decoded to determine which registers are being requested, what operation should be performed on those registers, and which register the result of the operation should be written into. Stage 3 is where the operation is actually executed onto the requested registers and the result is generated. Stage 4 is where the result is written back to the designated register. Each stage is separated by their interstage registers. Between Stage 3 and Stage 4 is a unit called “Data Forwarding” which will be discussed later.

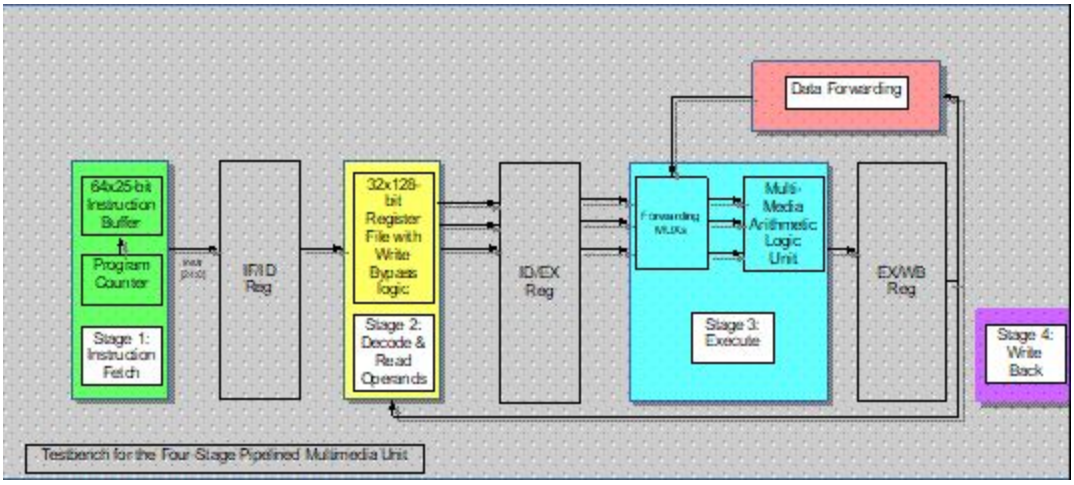

Figure 1. Four-Stage Pipelined Multimedia Unit Reference Image

Description of Stages

Stage 1 consists of a unit called “Instruction Buffer” along with its interstage register “IF/ID Register” which is shown in Figure 2.1. The Instruction Buffer unit was implemented as a behavioral model and can store up to 64 25-bit instructions. On each clock cycle, the instruction pointed by the PC is fetched and the PC is incremented by one to prepare for the next instruction. The interstage register is used to decode the fetched instruction and determine which registers are being requested and brings this information to stage two of the pipeline.

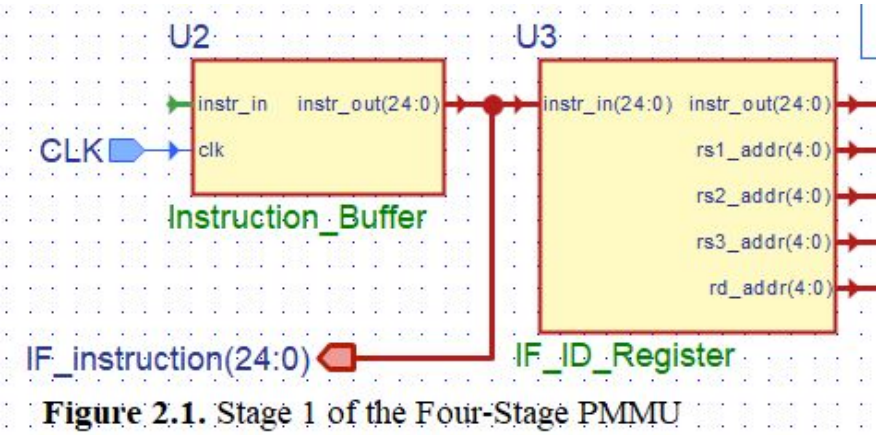

Figure 2.1. Stage 1 of the Four-Stage PMMU

Stage 2 consists of a unit called "Register File" along with its interstage register "ID/EX Register" which is shown in Figure 2.2. The Register File unit was implemented as a behavioral model and can hold up to 32 128-bit registers. On each clock cycle, the unit can read up to three registers and write up to one register. Writing to a register is determined by the signal "wr_enable". Reading and writing to registers occur at different halves of each clock cycle; every rising edge of the clock (first half of the clock cycle) will write to a register if applicable and every falling edge (second half of the clock cycle) will read up to three registers. The interstage register is used to decode what type of operation should be performed in the Arithmetic Logic Unit, or ALU, which is stage three of the pipeline.

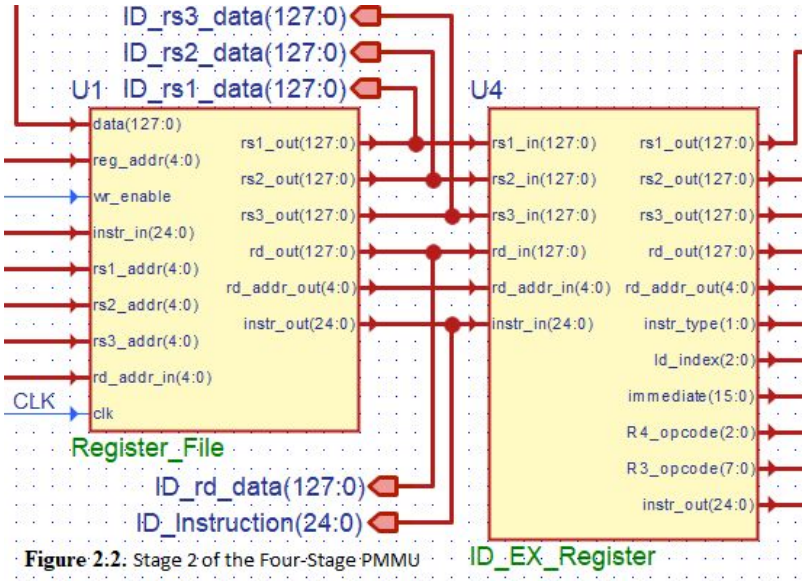

Figure 2.2. Stage 2 of the Four-Stage PMMU

Stage 3 consists of a unit called "Forwarding Mux", an ALU, and its interstage register "EX/WB Register" which is shown in Figure 2.3a. The Forwarding Mux unit is connected to the "Data Forwarding" unit which will be discussed later. The main function of the Forwarding Mux unit is to forward the most recent value of a register even if the value has not been written into

the Register File unit yet. Only valid data is considered for forwarding. The ALU performs the operations based on the opcodes that were decoded from its current instruction. The interstage register brings the result of the ALU into stage four of the pipeline.

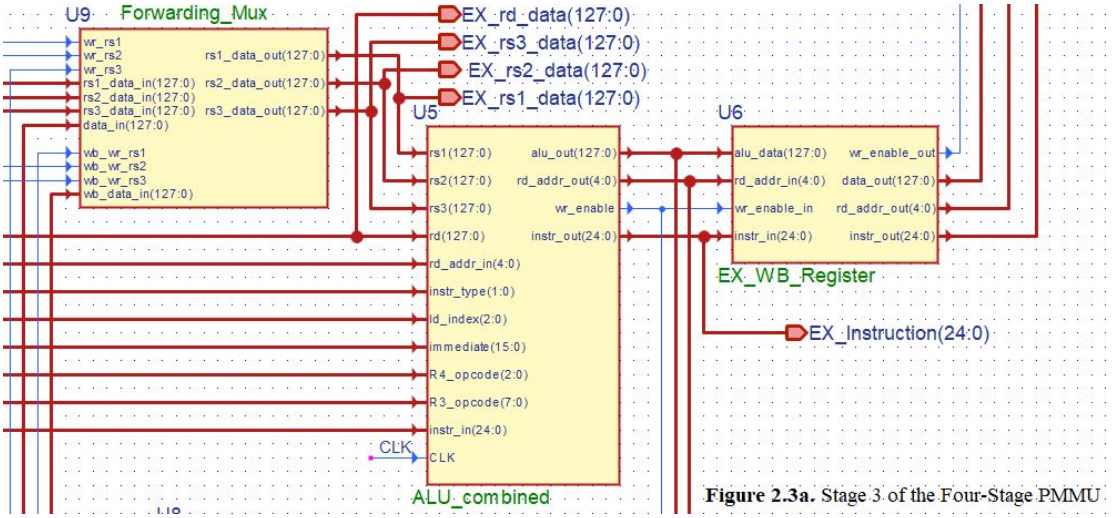

Figure 2.3a. Stage 3 of the Four-Stage PMMU

An inside look of the ALU is shown in Figure 2.3b, which separates the functions of the three instruction formats: Load Immediate (I-Type), R4-Type, and R3-Type. All operations for each type are performed, but the unit "ALU_select" determines which output is the correct value to be brought into its interstage register. Each component of the ALU was implemented as a behavioral model.

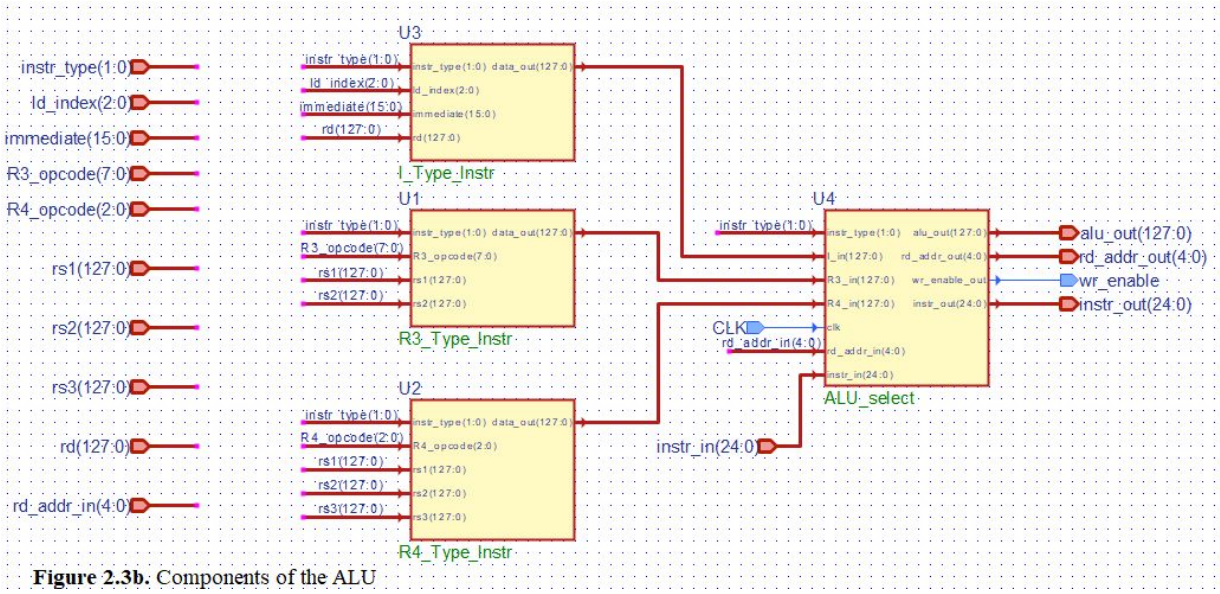

Figure 2.3b. Components of the ALU

Stage 4 consists of a unit called "Writeback" which can be seen in Figure 2.4. On each clock cycle, this unit will bring data from the EX/WB Register into the Register File along with the signal "WB_wr_enable" to determine if the data should be written into the Register File.

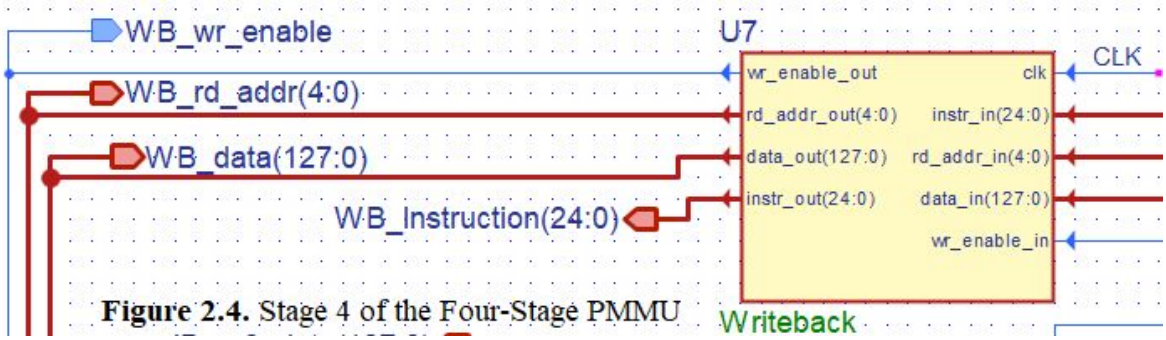

Figure 2.4. Stage 4 of the Four-Stage PMMU

The Data Forwarding unit's main function is to forward data that has been recently outputted from the ALU unit or is currently in the Writeback unit and has not been written into the Register File unit yet. The Data Forwarding unit takes the destination register address, or rd_address, from both the ALU and Writeback units and the instruction that is currently in Stage 2 of the pipeline is forwarded into the Data Forwarding unit. Each possible rd_address' of the instruction is compared to the rd_address of the ALU unit or the Writeback unit and if one of them matches, the data is forwarded into the Forwarding Mux unit along with a respective write signal. The Forwarding Mux unit checks the write signal to determine if the more recent value should be used before forwarding the data into the ALU for the current instruction. A code snippet for forwarding after the ALU stage is shown in Figure 2.5.

```
if wr_enable_in = '1' then
    if to_integer(unsigned(rd_addr_in)) = to_integer(unsigned(instr_in(19 downto 15))) then
        wr_rs3 <= '1';
    else
        wr_rs3 <= '0';
    end if;

    if to_integer(unsigned(rd_addr_in)) = to_integer(unsigned(instr_in(14 downto 10))) then
        wr_rs2 <= '1';
    else
        wr_rs2 <= '0';
    end if;

    if to_integer(unsigned(rd_addr_in)) = to_integer(unsigned(instr_in(9 downto 5))) then
        wr_rs1 <= '1';
    else
        wr_rs1 <= '0';
    end if;

    data_out <= data_in;
end if;
```

Figure 2.5. Code Snippet for Data Forwarding unit

The entire Four-Stage PMMU schematic is in the file "MMU_Schematic.pdf" which will be attached along with this report. The output signals are used to show where the testbench is grabbing its data from.

Instruction Types

There are three different types of instructions that the PMMU can execute: Load Immediate (I-Type), R4-Type, and R3-Type.

Figure 3.1. I-Type Format

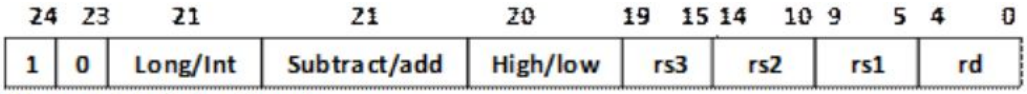

Figure 3.2. R4-Type Format

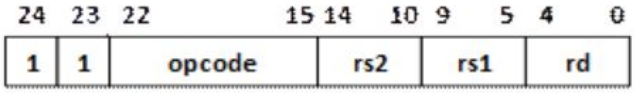

Figure 3.3. R3-Type Format

The system determines each instruction format by checking the first two MSB's, or bits 24 and 23. For I-Type instructions only bit 24 is needed so the system checks for either "00" or "01" while R4-Type instructions check for "10" and R3-Type instructions check for "11".

Saturation Implementation

In VHDL the range of the integer type is from [-2^{31}, 2^{31} - 1] and if overflow or underflow occurs after an operation, the value will "roll over". Consider the following: if the values (in hex) 7FFF FFFF and 0000 0001 were added the result would be 8000 0000. In decimal terms the expression is:

$$2147483647 + 1 = -2147483648.$$

We can conclude that if two positive 32-bit numbers were added together and the result became negative, then overflow occurred and the result needs to be saturated to 7FFF FFFF instead. Similarly, if two negative 32-bit numbers were added together and the result became positive, then underflow occurred and the result needs to be saturated to 8000 0000. This logic was used for operations such as R4-Type instructions with opcodes of 000 to 011, or signed integer addition/subtraction operations. A code snippet for a 32-bit number addition case is shown in Figure 4.1.

```
signed_product := to_integer(signed(rs3(31 downto 16))) * to_integer(signed(rs2(31 downto 16)));
signed_sum := signed_product + to_integer(signed(rs1(31 downto 0)));
if signed_sum < 0 and (rs1(31) = '0' and signed_product > 0) then
    -- overflow detected
    data_out(31 downto 0) <= (others => '1');
    data_out(31) <= '0';
elsif signed_sum > 0 and (rs1(31) = '1' and signed_product < 0) then
    -- underflow detected
    data_out(31 downto 0) <= (others => '0');
    data_out(31) <= '1';
else
    data_out(31 downto 0) <= std_logic_vector(to_signed(signed_product, 32));
end if;
```

Figure 4.1. Code snippet for overflow/underflow for 32-bit addition

R4-Type instructions with opcodes of 100 to 111, or signed long integer addition/subtraction operations, have a slight difference in their implementation of saturation, but works similarly as 32-bit numbers. Previously 32-bit numbers were converted to integer types but this method will not work for 64-bit numbers. Instead the MSB, or bit 64, of the 64-bit

number was checked to determine if overflow or underflow occurred. For example: when two positive 64-bit numbers are added (both their MSB's are '0') and the MSB of the sum is '1' then overflow has occurred and the result needs to be saturated. A code snippet for a subtraction case is shown in Figure 4.2.

```
product_sig := std_logic_vector(signed(rs3(31 downto 0)) * signed(rs2(31 downto 0)));
sum_sig := std_logic_vector(signed(rs1(63 downto 0)) - signed(product_sig));
if sum_sig(63) = '1' and (rs1(63) = '0' and product_sig(63) = '1') then
    --overflow detected
    data_out(63 downto 0) <= (others => '1');
    data_out(63) <= '0';
elsif sum_sig(63) = '0' and (rs1(63) = '1' and product_sig(63) = '0') then
    -- underflow detected
    data_out(63 downto 0) <= (others => '0');
    data_out(63) <= '1';
else
    data_out(63 downto 0) <= sum_sig;
end if;
```

Figure 4.2. Code snippet for overflow/underflow for 64-bit subtraction

An example of saturation occurring is the instruction at PC = 31. Since R22 was the destination of the preceding instruction, the value that was calculated for R22 is forwarded into the ALU for the current instruction. The low 32-bit value of each 64-bit value of R22 is multiplied to itself (basically squaring itself) to get a product of:

R22 = 7FFF 7FFF 7FFB 8002 7FFF 7FFF 7FFB 8002

Low 32-bit value of each 64-bit = 7FFB 8002 = 2147188738

$(2147188738)^2$ = 4610419476594032644 = 3FFB 8016 3FEE 0004 .

This value is then subtracted from each 64-bit value from R13 to get a difference of:

R13 = 8000 8004 8000 8004 8000 8004 8000 8004

Each 64-bit value = 8000 8004 8000 8004

64-bit product = 3FFB 8016 3FEE 0004

Difference = 1 4004 FFEE 4012 8000

and this difference requires 65-bits to represent. Since we only have 64-bit vectors, the MSB is cut out and we are left with a 64-bit value that appears to be positive since the MSB is now '0'. The underflow detection logic will essentially see:

negative - (positive) = positive

which doesn't make sense and will determine that the difference needs to be saturated to the smallest possible 64-bit signed number which is

8000 0000 0000 0000 .

The values that were used in the ALU as seen through the debugger is shown in Figure 4.3

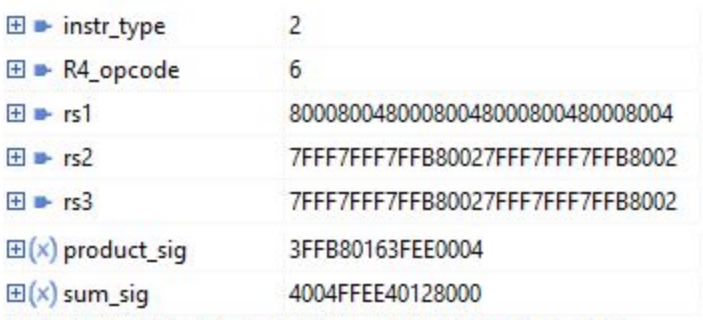

Figure 4.3. Debugging of instruction #31

Simulation Results

PC	Type	Instruction	Rd	Expected Register Value (Hex)
1	I	ldi R1, 0x7FFE index 0	R1	0000 0000 0000 0000 0000 0000 0000 7FFE
2	I	ldi R2, 0x0002 index 0	R2	0000 0000 0000 0000 0000 0000 0000 0002
3	R3	AHS R3, R1, R2	R3	0000 0000 0000 0000 0000 0000 0000 7FFF
4	R3	CLZ R4, R0, R31	R4	0000 0020 0000 0020 0000 0020 0000 0020
5	R3	A R5, R1, R4	R5	0000 0020 0000 0020 0000 0020 0000 801E
6	R3	AH R6, R1, R4	R6	0000 0020 0000 0020 0000 0020 0000 801E
7	R3	AND R7, R6, R5	R7	0000 0020 0000 0020 0000 0020 0000 801E
8	I	ldi R1, 0x7FFF index 1	R1	0000 0000 0000 0000 0000 0000 7FFF 7FFE
9	R3	BCW R3, R1, R0	R3	7FFF 7FFE 7FFF 7FFE 7FFF 7FFE 7FFF 7FFE
10	I	ldi R2, 0xFFFF index 1	R2	0000 0000 0000 0000 0000 0000 FFFF 0002
11	R3	BCW R4, R2, R0	R4	FFFF 0002 FFFF 0002 FFFF 0002 FFFF 0002
12	R3	ABSDB R5, R4, R2	R5	0001 0002 0001 0002 0001 0002 0000 0000
13	R3	MPYU R6, R5, R3	R6	0000 FFFC 0000 FFFC 0000 FFFC 0000 0000
14	R3	MSGN R7, R3, R5	R7	7FFF FFFF 7FFF FFFF 7FFF FFFF 0000 0000
15	R3	POPCNTW R9, R3, R31	R9	0000 001D 0000 001D 0000 001D 0000 001D
16	R3	DUPLICATED MPYU OPCODE	-----	NULL (wr_enable = 0)

17	R3	OR R8, R4, R0	R8	FFFF 0002 FFFF 0002 FFFF 0002 FFFF 0002
18	R3	ROT R10, R1, R2	R10	8000 0000 0000 0000 0000 0000 1FFF DFFF
19	R3	ROTW R11, R1, R2	R11	0000 0000 0000 0000 0000 0000 9FFF DFFF
20	R3	SHLHI R12, R4, R2	R12	FFFC 0008 FFFC 0008 FFFC 0008 FFFC 0008
21	R3	SFH R13 R3, R4	R13	8000 8004 8000 8004 8000 8004 8000 8004
22	R3	SFW R14, R6, R4	R14	FFFE 0006 FFFE 0006 FFFE 0006 FFFF 0002
23	R3	SFHS R15, R6, R5	R15	0001 0006 0001 0006 0001 0006 0000 0000
24	R3	NAND R16, R0, R0	R16	FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
25	R4	000 R17, R3, R13, R13	R17	7FFF FFFF 7FFF FFFF 7FFF FFFF 7FFF FFFF
26	R4	001 R18, R5, R3, R5	R18	0001 8001 0001 8001 0001 8001 0000 0000
27	R4	010 R19, R5, R1, R1	R19	0001 0002 0001 0002 0001 0002 C001 FFFC
28	R4	011 R20, R5, R1, R1	R20	0001 0002 0001 0002 0001 0002 C000 FFFF
29	R4	100 R21, R3, R1, R1	R21	7FFF 7FFE 7FFF 7FFE 7FFF FFFF FFFF FFFF
30	R4	101 R22, R3, R8, R8	R22	7FFF 7FFF 7FFB 8002 7FFF 7FFF 7FFB 8002
31	R4	110 R23, R13, R22, R22	R23	8000 0000 0000 0000 8000 0000 0000 0000
32	R4	111 R24, R23, R16, R16	R24	8000 0000 0000 0000 8000 0000 0000 0000
33	R3	NOP	----	NULL (wr_enable = 0)

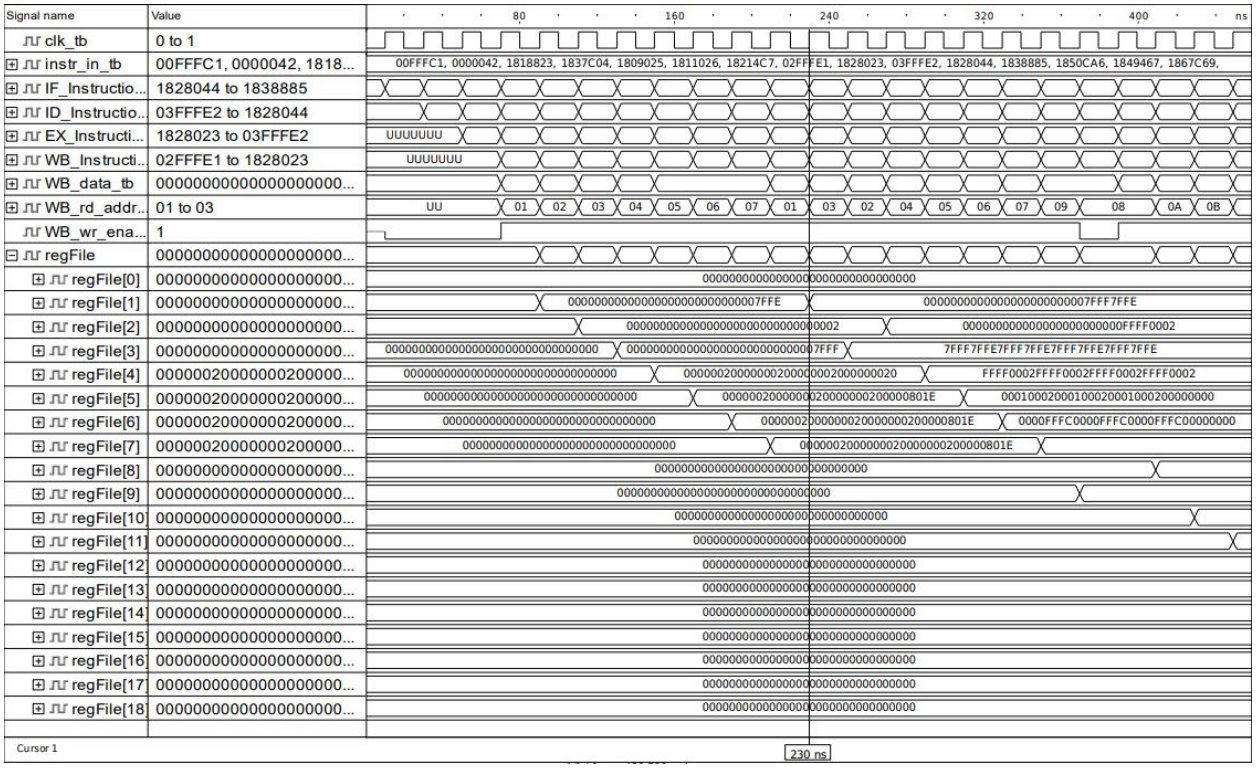

Figure 5.1. Simulation Results from PC = 1 to PC = 7

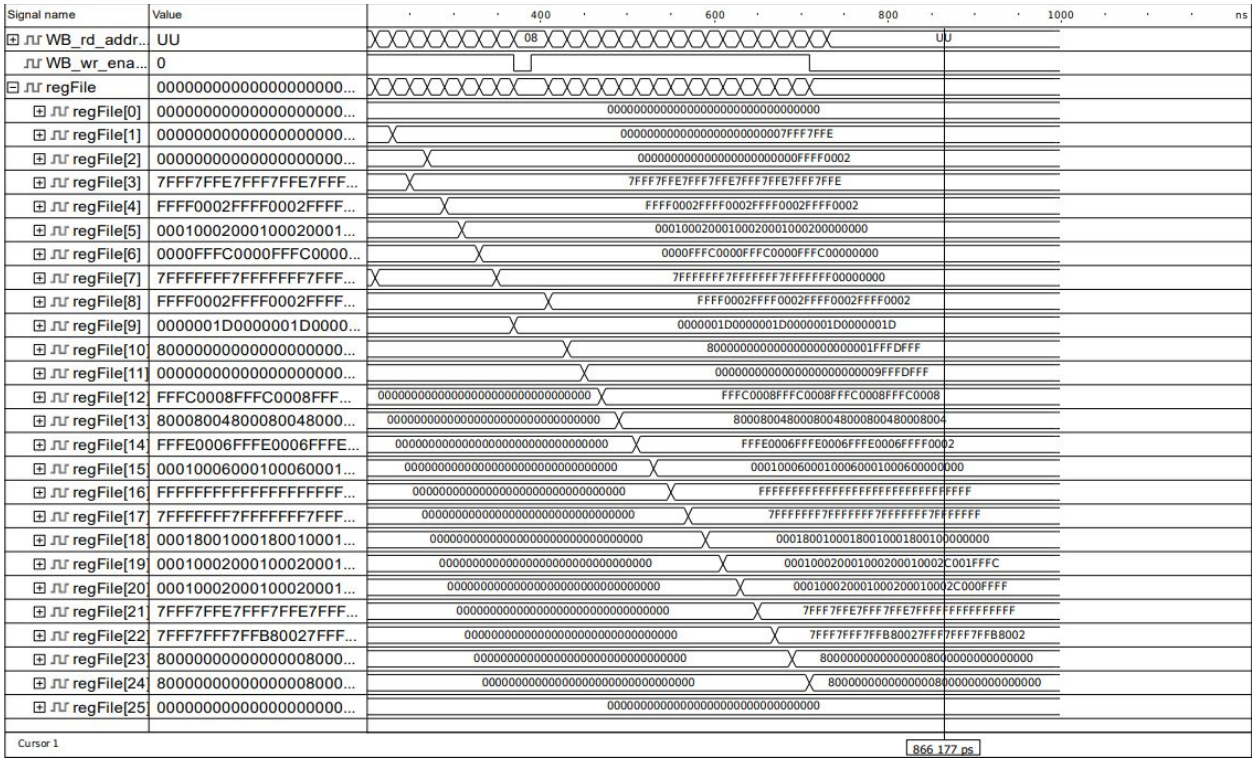

Figure 5.2. Simulation Results from PC = 8 to PC = 33

Conclusion

The four-stage PMMU is functioning as intended. Each instruction takes four clock cycles to execute. A forwarding unit was used in cases of data hazards instead of having to stall the pipeline. A custom assembler was designed in C++ and was used to encode the instructions from text into binary format. A testbench was used to generate some of the signals during each stage of the four-stage PMMU into a separate text file.