4 Stage Pipelined Multimedia Unit Design with VHDL

ESE 345

Fall 2024

Professor Mikhail Dorojevets

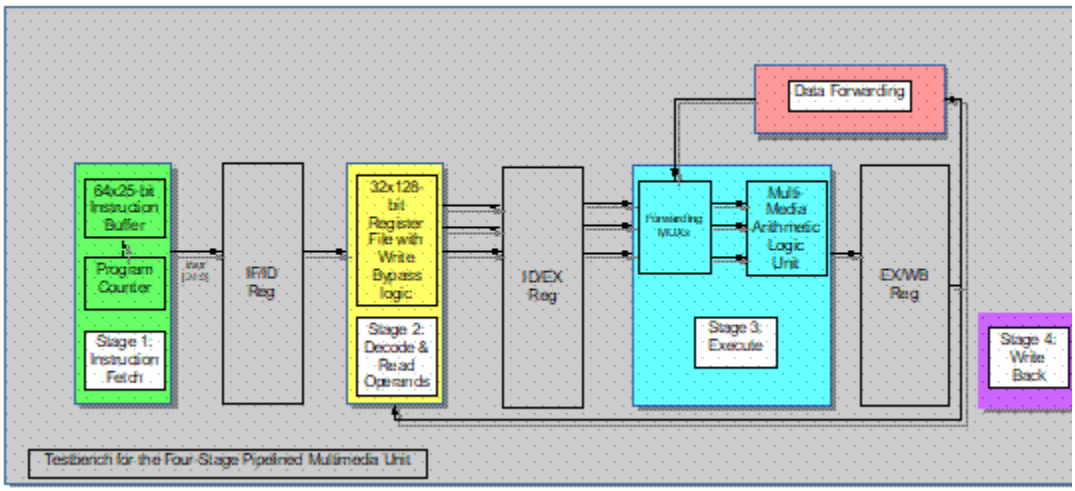
Naafiul Hossain and Muhammed Sharjeel

Introduction and Goal:

ESE 345, *Computer Architecture*, is a course at Stony Brook University that focuses on processor design, memory structure, and advanced system design concepts. The course culminates in a capstone project: designing a pipelined multimedia processing unit using a hardware description language of our choice–in this case VHDL. Additionally, we were tasked with developing a custom assembler program tailored to our hardware design–C++.

The final evaluation required us to test, validate, and present our work to a teaching assistant. During the presentation, we demonstrated our assembler's functionality by converting a MIPS-style assembly program—provided by the TA—into binary machine code. We showcased the execution of the code and highlighted the memory addresses where results were stored.

The project is broken into eight key units as shown in this illustration:



Multimedia ALU: Takes up to three inputs from the Register File, and calculates the result based on the current instruction to be performed. The ALU must be implemented as behavioral model in VHDL or continuous assignment (dataflow models in Verilog).

Register File: The register file has 32 128-bit registers. On any cycle, there can be 3 reads and 1 write. When executing instructions, each cycle two/three 128-bit register values are read, and one 128-bit result can be written if a write signal is valid. This register write signal must be explicitly declared so it can be checked during simulation and demonstration of your design. The register module must be implemented as a behavioral model in VHDL (dataflow/RTL model in Verilog).

Instruction Buffer: The instruction buffer can store 64 25-bit instructions. The contents of the buffer should be loaded by the testbench instructions from a test file at the start of simulation. On each cycle, the instruction specified by the Program Counter (PC) is fetched, and the value of PC

is incremented by 1. The Instruction Buffer module must be implemented as a behavioral model in VHDL (dataflow/RTL model in Verilog).

Forwarding Unit: Every instruction must use the most recent value of a register, even if this value has not yet been written to the Register File. Be mindful of the ordering of instructions; the most recent value should be used, in the event of two consecutive writes to a register, followed by a read from that same register. Your processor should never stall in the event of hazards. Take extra care of which instructions require forwarding, and which ones do not. Namely, NOP and the instructions with Immediate fields do not contain one/two register sources. Only valid data and source/destination registers should be considered for forwarding.

Four-Stage Pipelined Multimedia Unit: Clock edge-sensitive pipeline registers separate the IF, ID, EXE, and WB stages. Data should be written to the Register File after the WB Stage. All instructions (including li) take four cycles to complete. This pipeline must be 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.

Testbench: This module loads the instruction buffer using data loaded from a file, begins simulation, and upon completion, compares the contents of the register file to a file containing the expected results. This expected results file does not need to be auto-generated. Instead, this can be manually entered when designing a test program. This must be implemented as a behavioral model.

Assembler: This is a separate program written in any language your team prefers (i.e. Java, C++, Python). Its purpose is to convert an assembly file to the binary format for the Instruction Buffer. This assembler does not need to be robust, and can assume very specific syntax rules that you as a team decide.

Results File: This file must show the status of the pipeline for each cycle during program execution. It should include the opcodes, input operand, and results of the execution of instructions, as well as all relevant control signals and forwarding information. This should be carried out by your testbench.

Instruction Format

The instruction format for the Multimedia Unit is designed to support three distinct types of instructions: I-type, R3-type, and R4-type, each with a unique structure. I-type instructions are 24 bits long and are identified by a 1 in the 23rd bit. They include a 16-bit immediate value (bits 20–5) that is loaded into the

rd register specified by bits 4–0. Additionally, bits 22–21 indicate the byte of the rd register where the immediate value will be placed. This format allows for efficient handling of immediate data operations.

R3-type instructions, also 24 bits, are characterized by the 00 in bits 23–22. These instructions include a 7-bit opcode (bits 21–15) that specifies the operation to be performed, and three registers: rs2, rs1, and rd, represented by bits 14–10, 9–5, and 4–0, respectively. This format supports operations that require two source registers and one destination register, making it suitable for arithmetic and logical computations. The clear structure ensures accurate decoding and execution within the pipeline.

R4-type instructions expand on the R3 format by adding a field for additional functionality. Identified by 01 in bits 23–22, this format includes MA/MS/I/h bits in positions 21–20 to select specific operations. It uses four registers: rs3 (bits 19–15), rs2 (bits 14–10), rs1 (bits 9–5), and rd (bits 4–0), enabling more complex instructions. The structured approach across these three formats allows the pipeline to efficiently process diverse instruction sets, optimizing performance and reducing execution time.

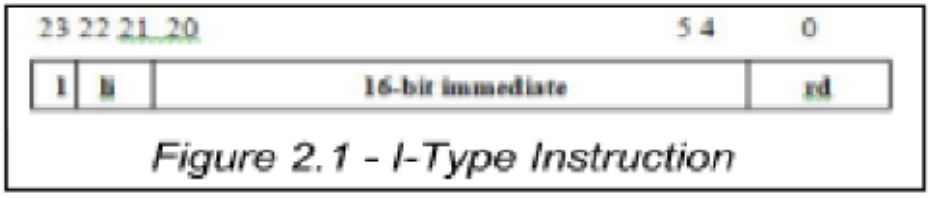


Figure 2.1 - I-Type Instruction

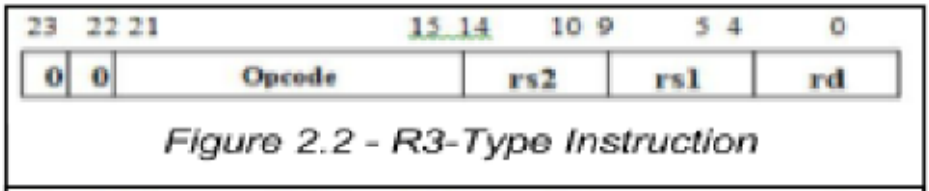


Figure 2.2 - R3-Type Instruction

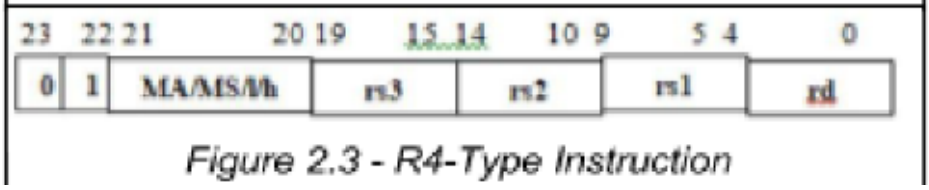


Figure 2.3 - R4-Type Instruction

Logic Implementation:

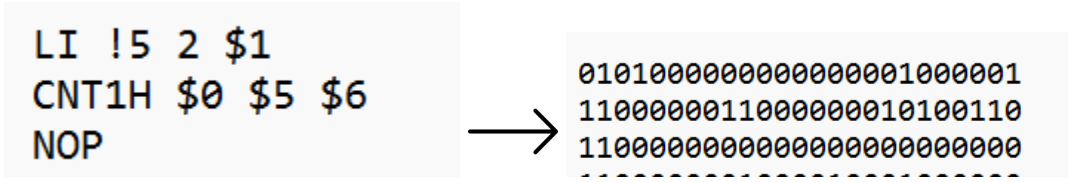
The system implementation involved several critical components working in harmony to create a functional Pipelined Multimedia Unit (PMU). The testbench played a central role, programming the instruction buffer, providing clock and reset signals, and validating results by comparing the processor's memory with expected outputs.

It operated in three states (resetting, program control, and normal operation) feeding instructions into program memory during the programming state and generating debug messages for any discrepancies. The instruction buffer managed up to 64 instructions, propagating instruction flags and signaling when the program was complete. It ensured proper sequencing of instructions through pipelining stages, using a forwarding register for spacing and accurate handoff of instruction data.

The register file handled data storage and retrieval for arithmetic operations while mitigating hazards through write-back checks. It outputs data based on instruction types, supporting seamless integration with the pipeline. Forwarding registers in the instruction fetch and decode stages ensured proper data flow, maintaining pipeline spacing and forwarding valid instruction flags. Throughout, verification strategies emphasized transparency and accuracy, leveraging step-by-step memory checks against expected results. This robust design, paired with effective debugging and validation, ensured the PMU met its functional and performance objective

Assembler:

Our program, written in C++, is a MIPS assembly to machine code translator. It reads MIPS assembly instructions from an input file (assemblerN.txt), converts each instruction or register to its corresponding machine code using a predefined mapping stored in a std::map, and writes the resulting machine code to an output file (Nmachine_code.txt). The code includes a function to translate decimal numbers into 16-bit binary for immediate values and another function to map assembly instructions or registers to machine code strings. Each line in the input file is processed word by word: numeric values are converted to binary, and assembly instructions or registers are matched in the map for translation. The resulting machine code for each line is padded to 25 characters before being written to the output file. The program uses file handling, string manipulation, and the STL (Standard Template Library) to achieve efficient translation.

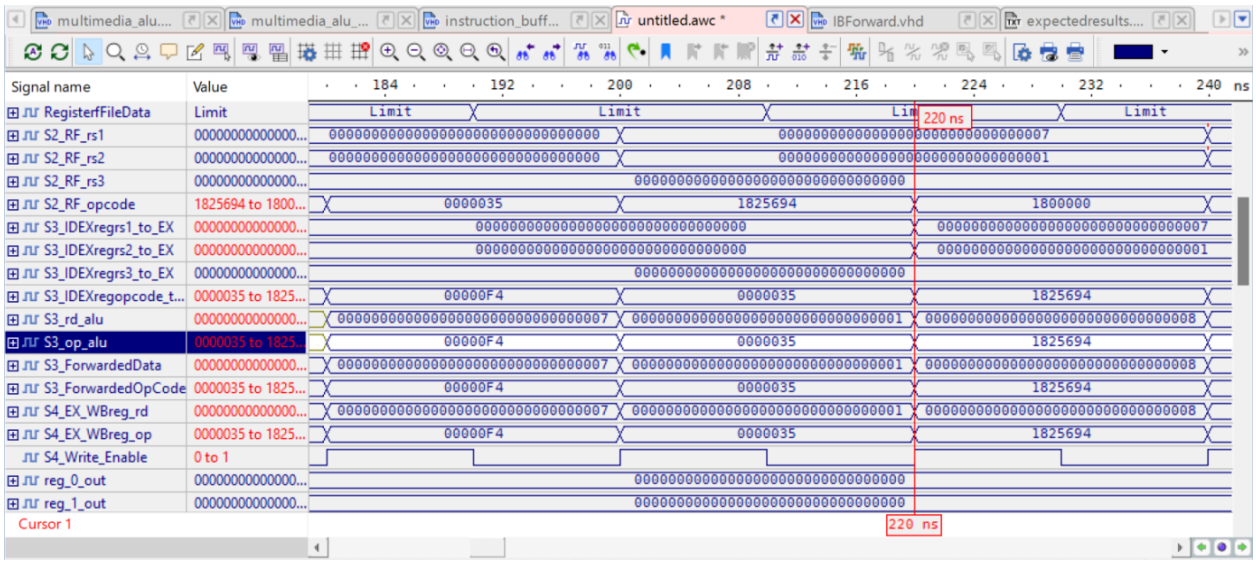


Forwarding:

The forwarding logic in the execute module is implemented by dynamically comparing specific fields of the current instruction’s opcode (EXin_opcode) with the forwarded instruction's opcode (opcode_fwd). It identifies data hazards by matching relevant bits and uses the forwarded result (rd_fwd) to supply the appropriate operand (rs1, rs2, or rs3). The logic categorizes hazards into three types: **LI hazards**, which focus on the lower bits of the opcode to detect dependencies on rs1; **R4 hazards**, which resolve dependencies for multi-operand instructions by comparing higher-order opcode bits with all three operands; and **R3 hazards**, which handle simpler two-operand instructions by selectively forwarding to rs1 or rs2. If a match is found and the forwarded instruction is valid (Valid_Instruction_In_FWD = '1'),

the corresponding operand is replaced with rd_fwd. If no hazard is detected, the inputs pass through to the outputs unchanged, ensuring seamless operation.

This logic works because it resolves data dependencies dynamically during the execution stage of a pipeline, eliminating the need for stalls. By forwarding results directly from a later stage to an earlier stage, the module avoids delays caused by waiting for register writes. This implementation is efficient and ensures the pipeline continues executing instructions without interruption. The decision-making process is driven by specific opcode patterns, which precisely identify when and where forwarding is needed. This mechanism works because it leverages the structure of pipelined execution, where instructions in different stages have predictable dependencies, allowing forwarding to mitigate data hazards without additional complexity.

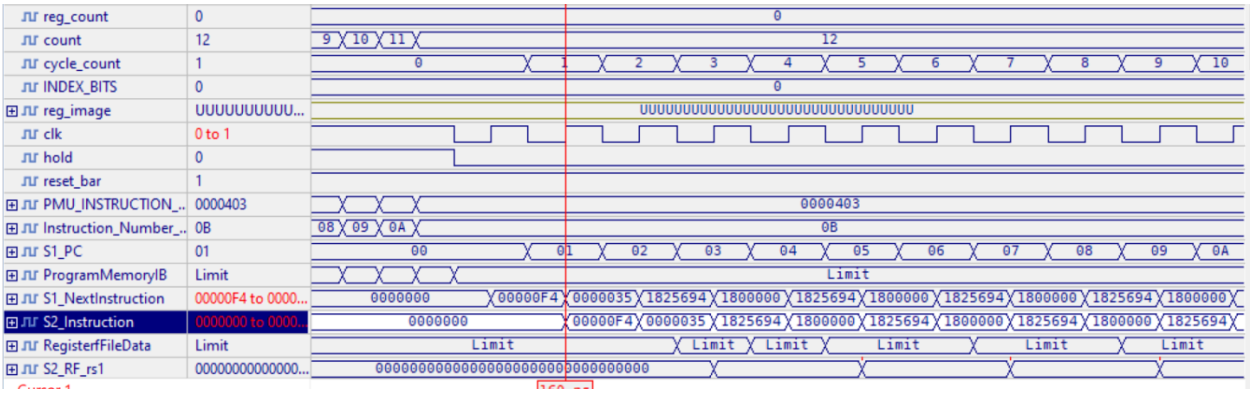


(The imagine above shows data forwarding in our Pipeline: This waveform diagram illustrates the forwarding of operand values (highlighted in the S3_ForwardedData signal) from later stages back to the Execute stage to resolve data dependencies, demonstrating how operand values are reused immediately in subsequent operations to avoid pipeline stalls and maintain computational efficiency)

Four instructions-PIPELINE

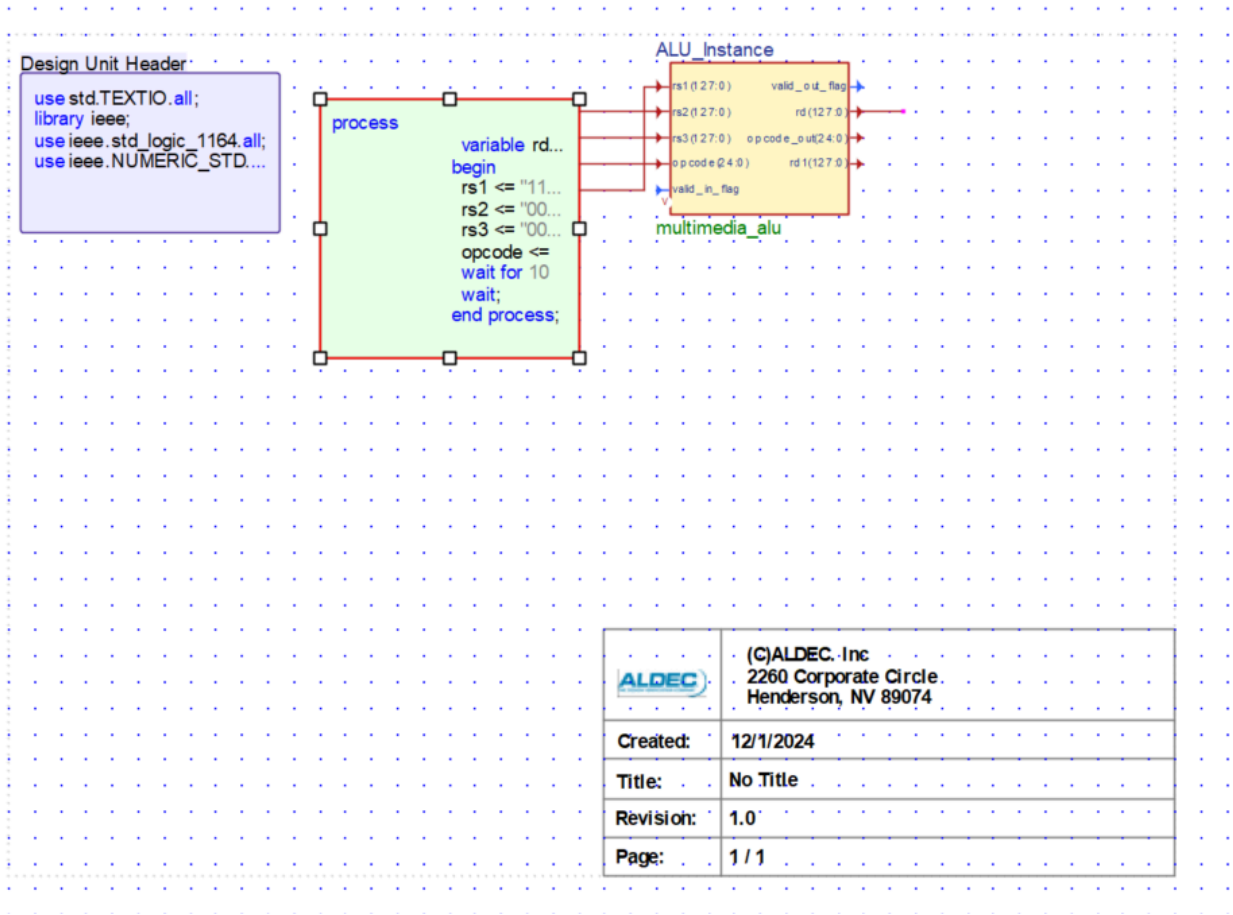
In this waveform that we generated, we're looking at a snapshot of a microprocessor's pipelining process, which allows multiple instructions to be processed simultaneously at different stages. Here, pipelining is illustrated through the S1_PC and S1_NextInstruction signals, showing where in the pipeline instructions are currently positioned. The S1_PC indicates the program counter for the instruction being processed, and S1_NextInstruction represents the address of the next instruction to be fetched. Meanwhile, the S2_Instruction seems to be idle at this point, indicating that it's either waiting for new instructions to

arrive from the fetch stage or there is a stall in the pipeline. This simultaneous handling of instructions at different stages optimizes throughput and efficiency, as while one instruction is being decoded, another can be fetched, reducing idle times and increasing the overall speed of the processing.

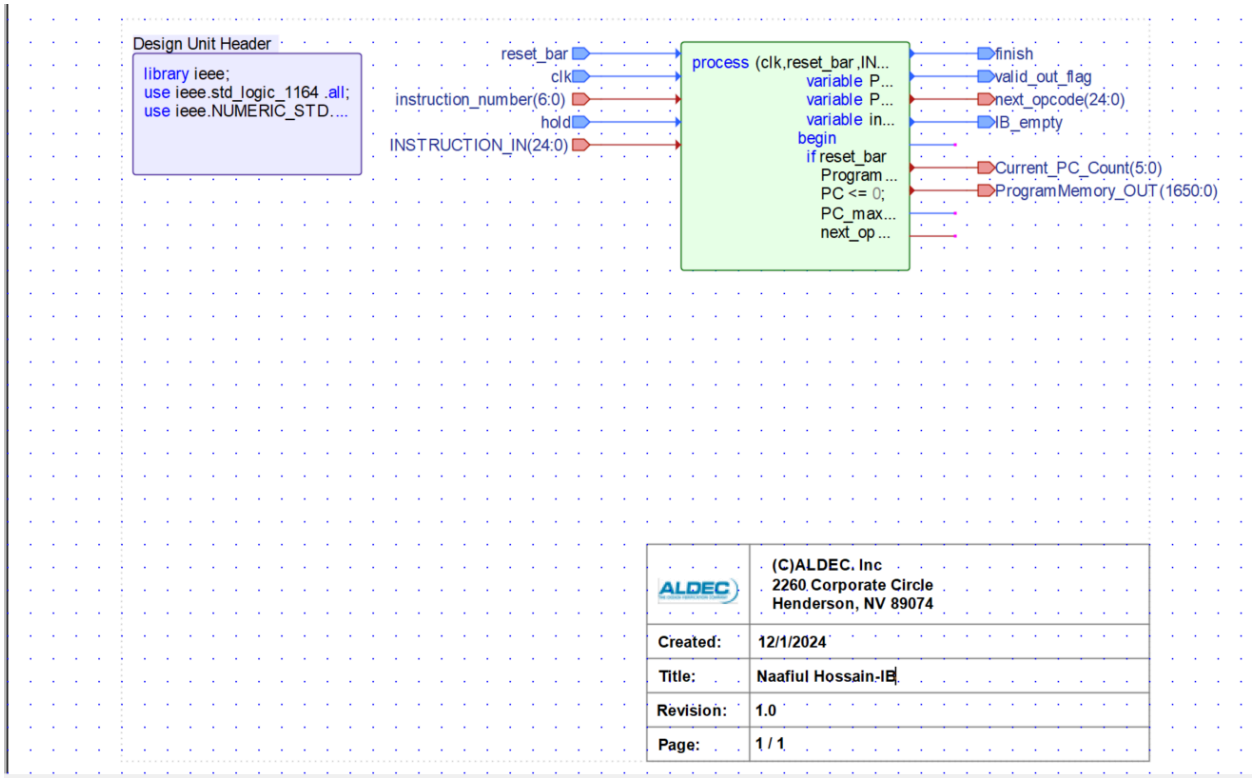


Block Diagrams

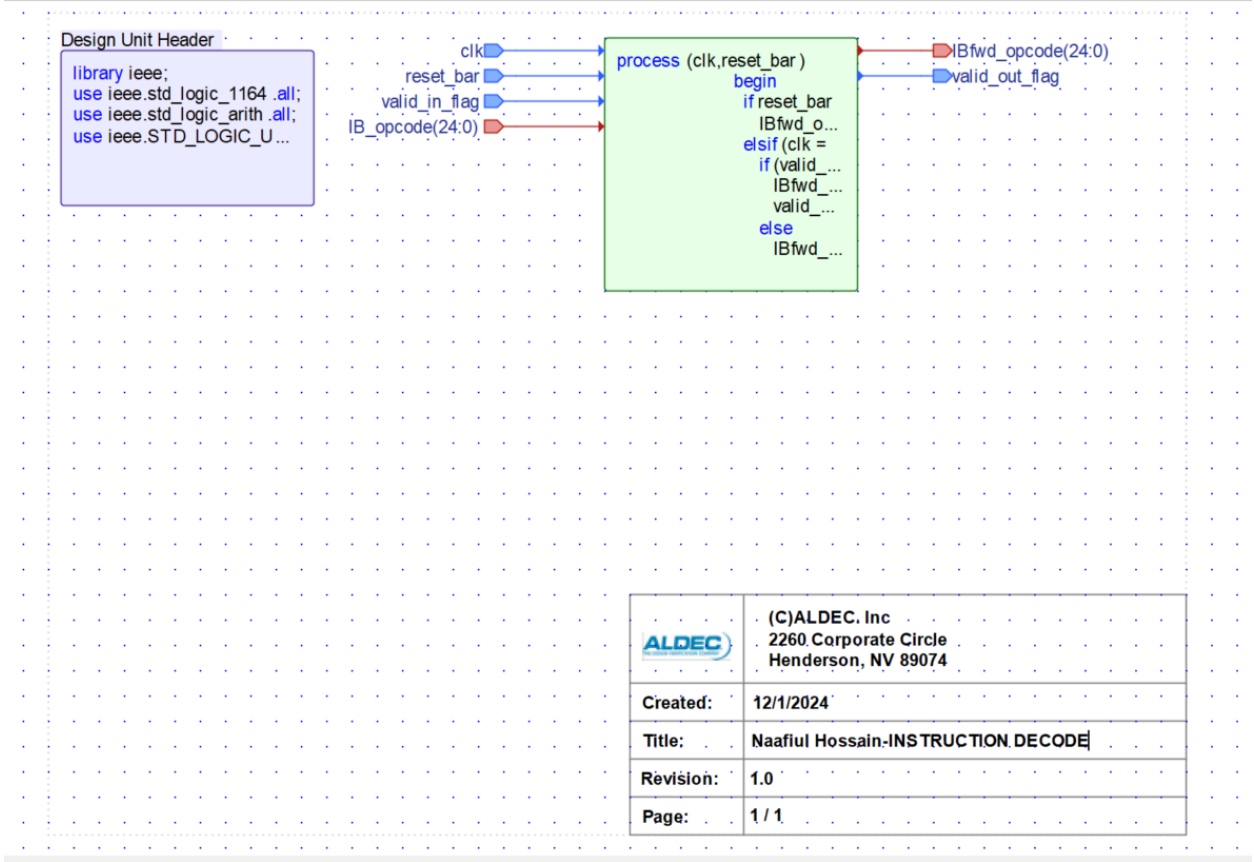
Alu:



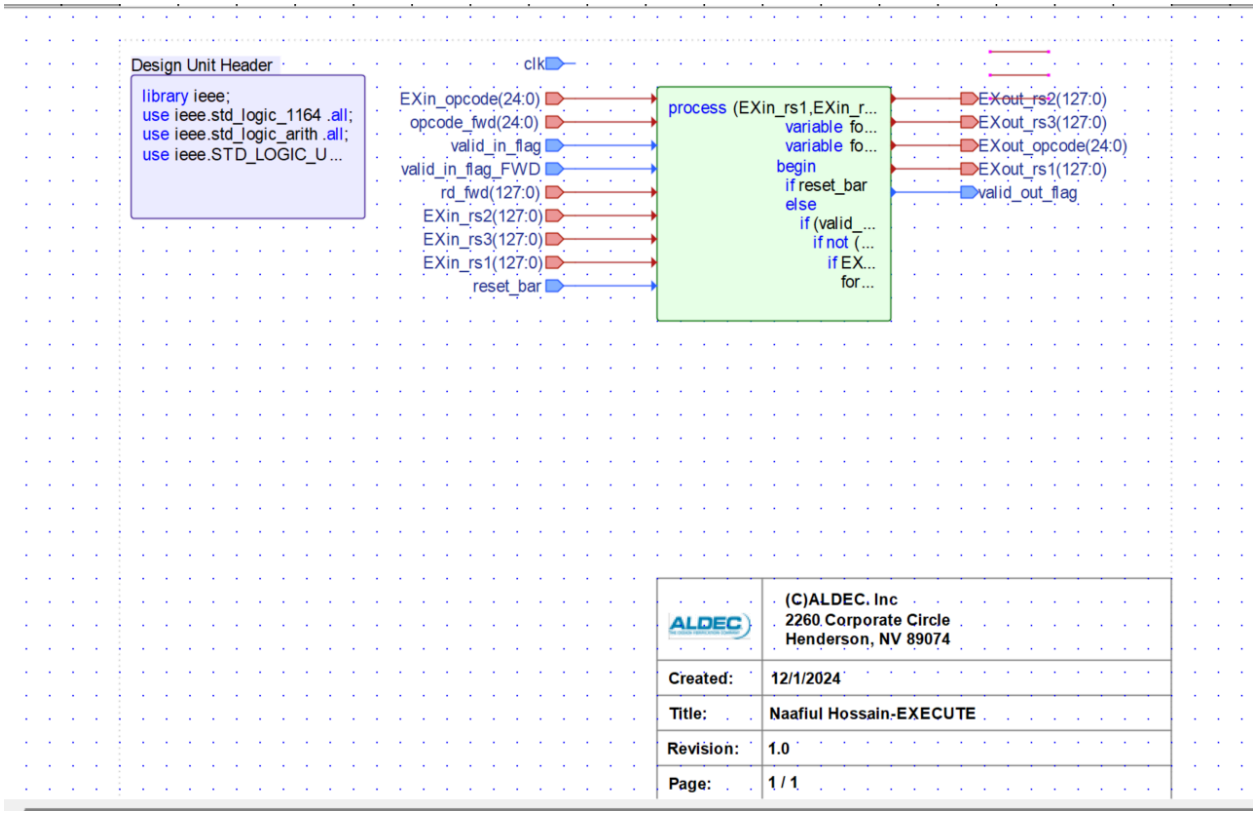
Instruction Buffer:



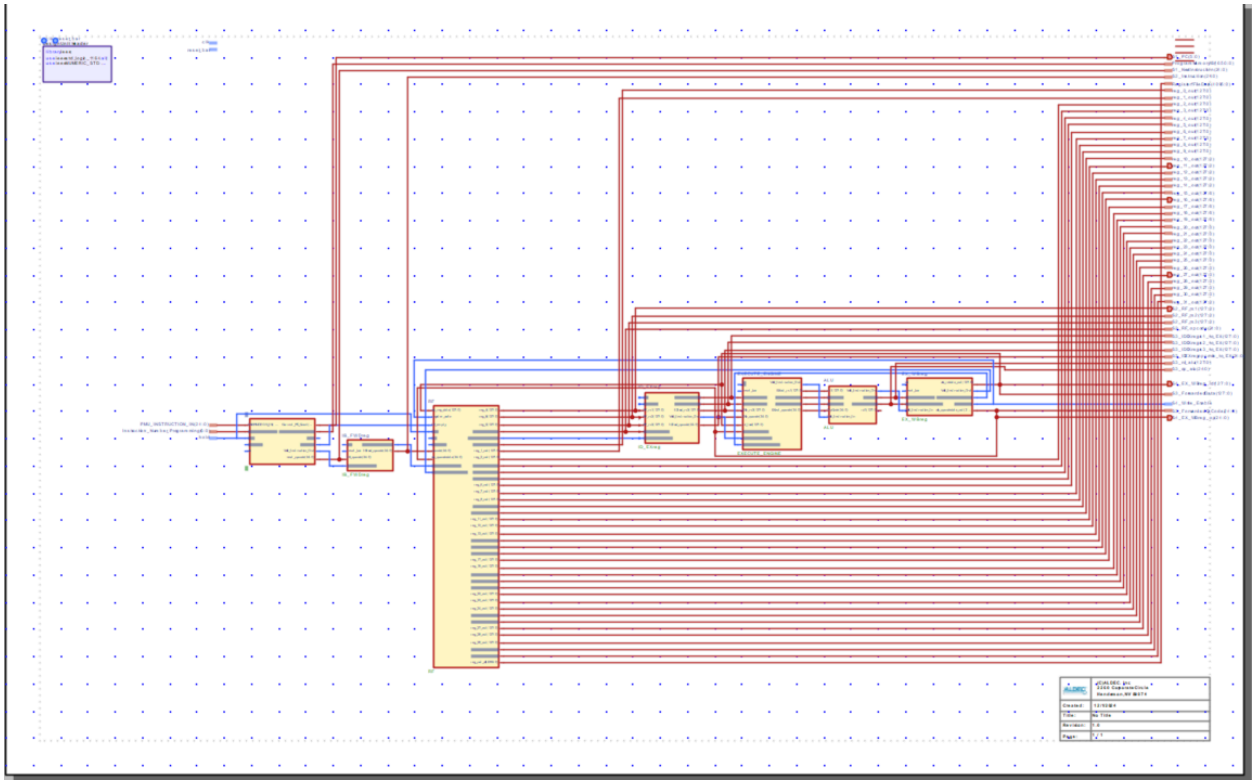
Instruction Decode:

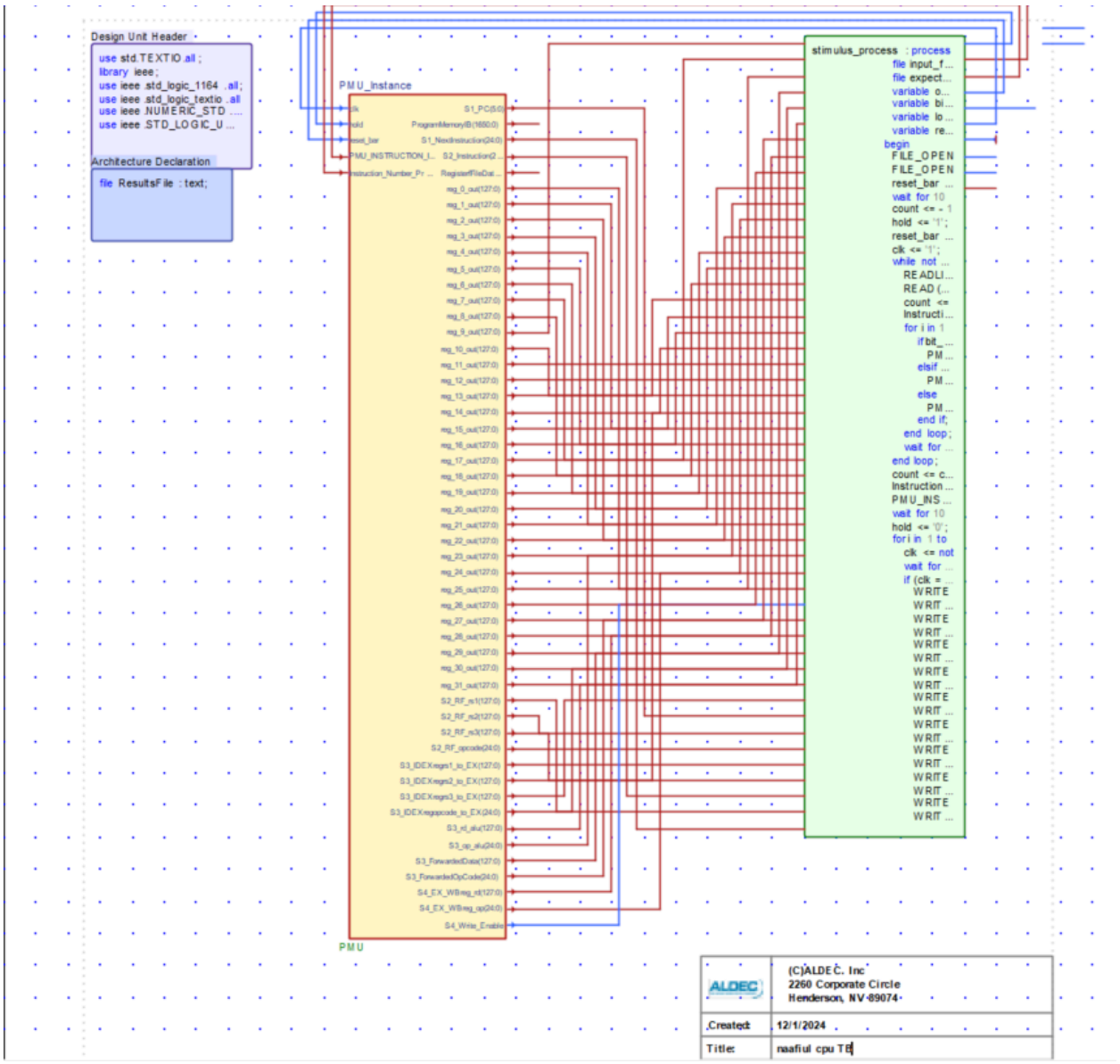


Execute:

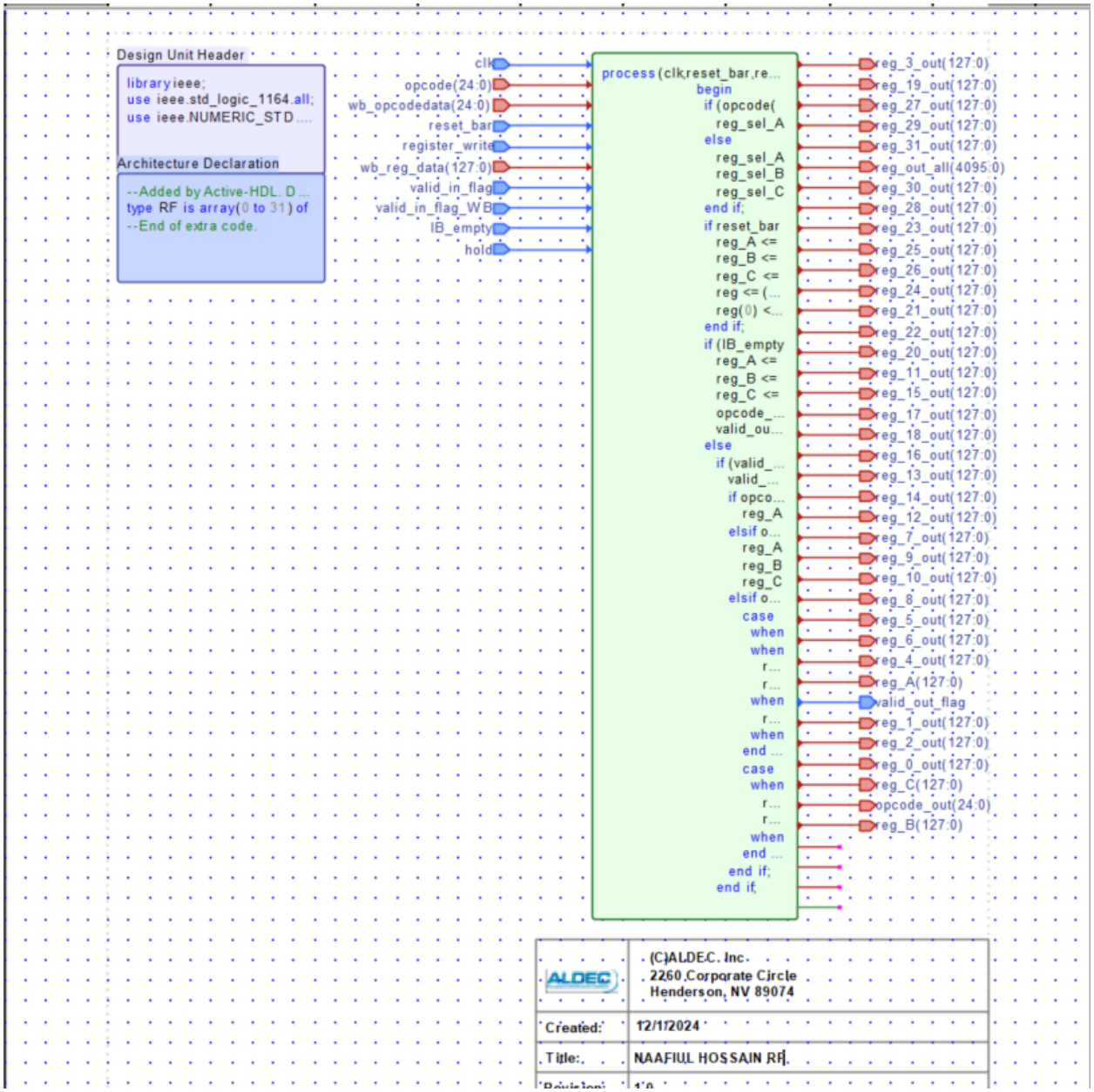


CPU/PMU/Central control unit:



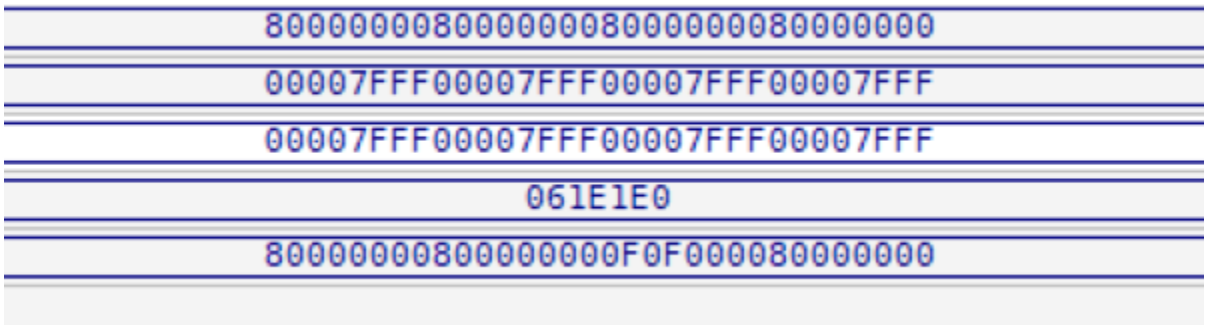

Design Unit Header
use std.TEXTIO.all ;
library ieee ;
use ieee .std_logic_1164 .all;
use ieee .std_logic_textio .all
use ieee .NUMERIC_STD ....
use ieee .STD_LOGIC_U ...
Architecture Declaration
file ResultsFile : text;
PMU_Instance
PMU
stimulus_process : process
file input_f...
file expect...
variable o...
variable bi...
variable lo...
variable re...
begin
FILE_OPEN
FILE_OPEN
reset_bar ...
wait for 10
count <= ...
hold <= '1';
reset_bar ...
clk <= '1';
while not ...
READLI...
READ (...
count <=
Instructi...
for i in 1
if bit_...
PM...
elsif ...
PM...
else
PM...
end if;
end loop;
wait for ...
end loop;
count <= c...
Instruction...
PMU_INS...
wait for 10
hold <= '0';
for i in 1 to
clk <= not
wait for ...
if (clk =
WRITE
WRIT ...
(C)ALDEC, Inc
2260 Corporate Circle
Henderson, NV 89074
Created: 12/1/2024
Title: naafiul cpu TB

RF:

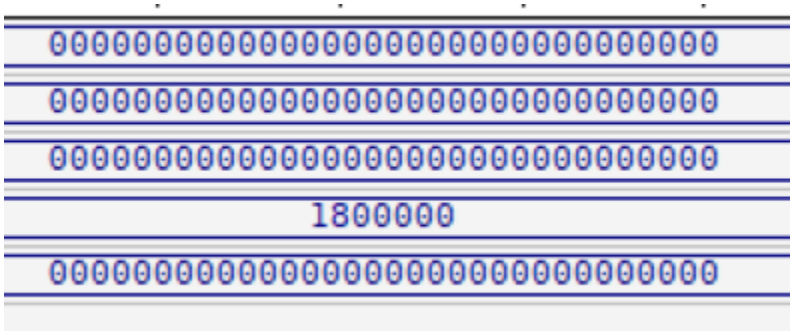

Design Unit Header
library ieee;
use ieee.std_logic_1164.all;
use ieee.NUMERIC_STD....
Architecture Declaration
--Added by Active-HDL. D...
type RF is array(0 to 31) of
--End of extra code.
clk
opcode(24:0)
wb_opcodedata(24:0)
reset_bar
register_write
wb_reg_data(127:0)
valid_in_flag
valid_in_flag_WB
IB_empty
hold
process (clk,reset_bar,re...
begin
if (opcode(
reg_sel_A
else
reg_sel_A
reg_sel_B
reg_sel_C
end if;
if reset_bar
reg_A <=
reg_B <=
reg_C <=
reg <= (...
reg(0) <...
end if;
if (IB_empty
reg_A <=
reg_B <=
reg_C <=
opcode_...
valid_ou...
else
if (valid_...
valid_...
if opco...
reg_A
elsif o...
reg_A
reg_B
reg_C
elsif o...
case
when
when
r...
r...
when
r...
when
end ...
case
when
r...
r...
when
end ...
end if;
end if;
reg_3_out(127:0)
reg_19_out(127:0)
reg_27_out(127:0)
reg_29_out(127:0)
reg_31_out(127:0)
reg_out_all(4095:0)
reg_30_out(127:0)
reg_28_out(127:0)
reg_23_out(127:0)
reg_25_out(127:0)
reg_26_out(127:0)
reg_24_out(127:0)
reg_21_out(127:0)
reg_22_out(127:0)
reg_20_out(127:0)
reg_11_out(127:0)
reg_15_out(127:0)
reg_17_out(127:0)
reg_18_out(127:0)
reg_16_out(127:0)
reg_13_out(127:0)
reg_14_out(127:0)
reg_12_out(127:0)
reg_7_out(127:0)
reg_9_out(127:0)
reg_10_out(127:0)
reg_8_out(127:0)
reg_5_out(127:0)
reg_6_out(127:0)
reg_4_out(127:0)
reg_A(127:0)
valid_out_flag
reg_1_out(127:0)
reg_2_out(127:0)
reg_0_out(127:0)
reg_C(127:0)
opcode_out(24:0)
reg_B(127:0)
ALDEC
(C)ALDEC, Inc.
2260 Corporate Circle
Henderson, NV 89074
Created:
12/17/2024
Title:
NAAFIUL HOSSAIN RF

Waveform Stimulations:

Load:



NOP:



SLHI: (ex. shift 6)

```
0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F
00007FFF00007FFF00007FFF00007FFF
00007FFF00007FFF00007FFF00007FFF
1809820
C3C0C3C0C3C0C3C0C3C0C3C0C3C0C3C0
```

AU:

```
FFFF0001F1F1F1747474748989898989
FF00FFFF0011000F00F0001000001010
00000000000000000000000000000000
1810000
FF000000F202F18375647499898999999
```

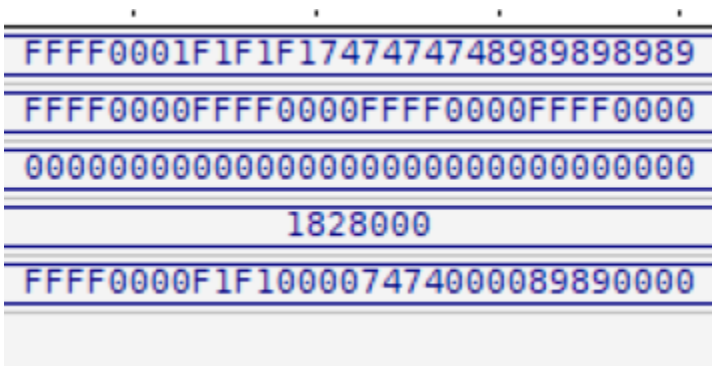
CNT1H:

```
FFFF0001F1F1F1747474748989898989
00000000000000000000000000000000
00000000000000000000000000000000
1818000
00100001000A000900080007000600006
```

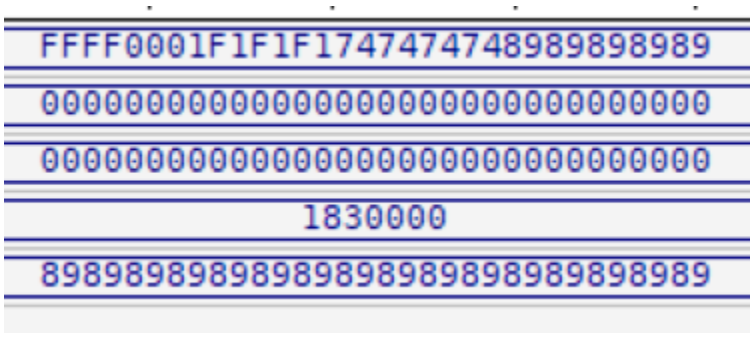
AHS:

```
FFFF0001F1F1F1747474748989898989
FF00FFFF0011000F00F0001000001010
00000000000000000000000000000000
1820000
FEFF0000F202F18375647499898999999
```

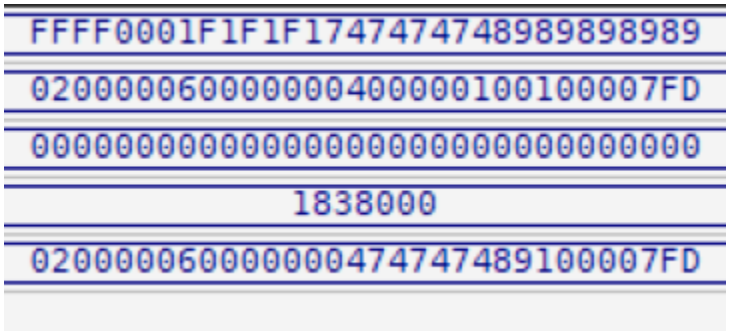
AND:



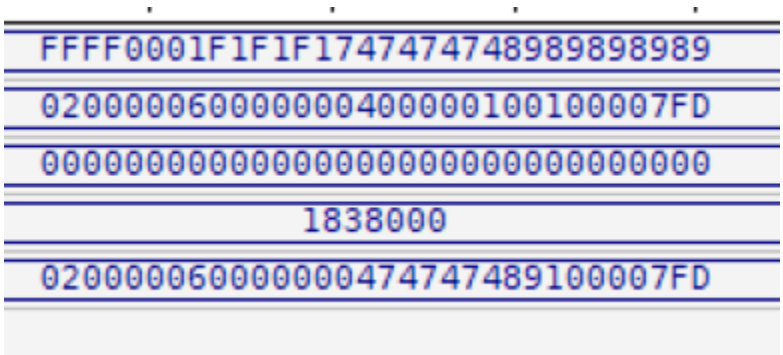
BCW:



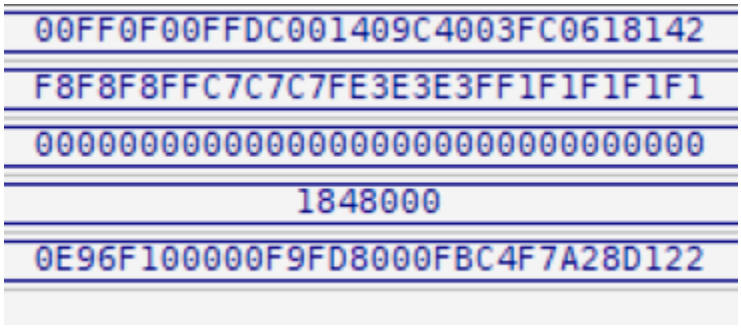
MAXWS:



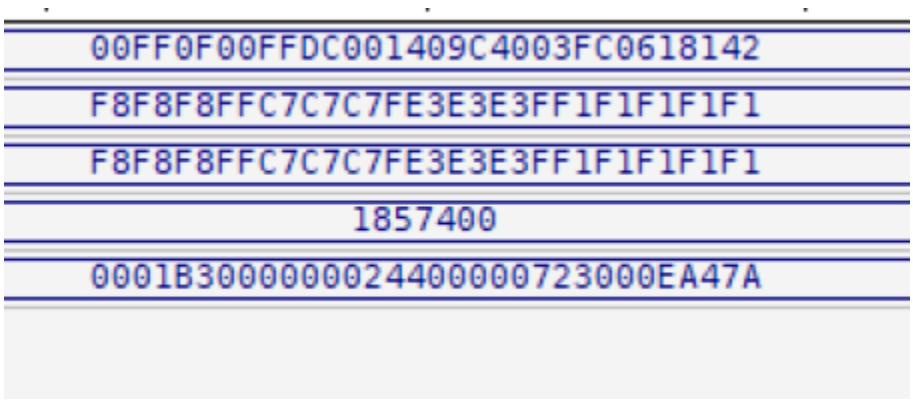
MINWS:



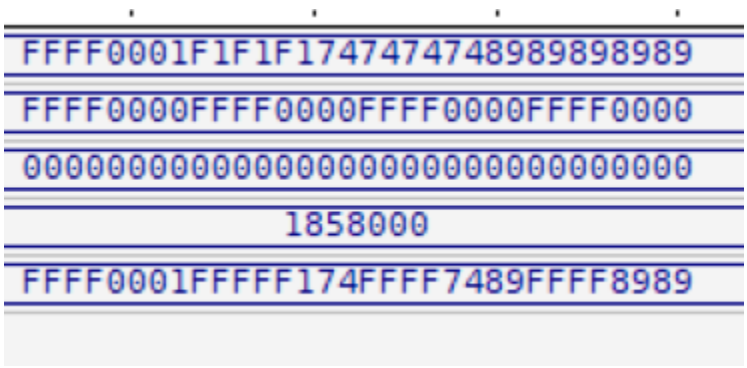
MLHU:



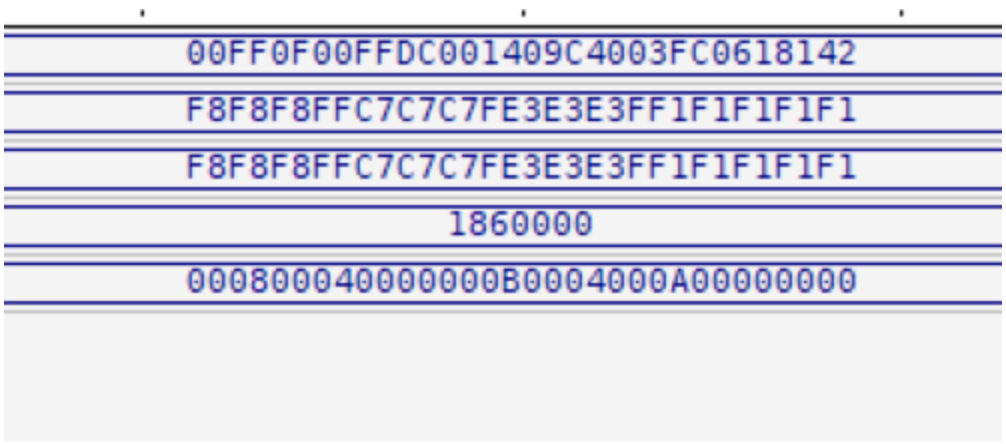
MLHCU:



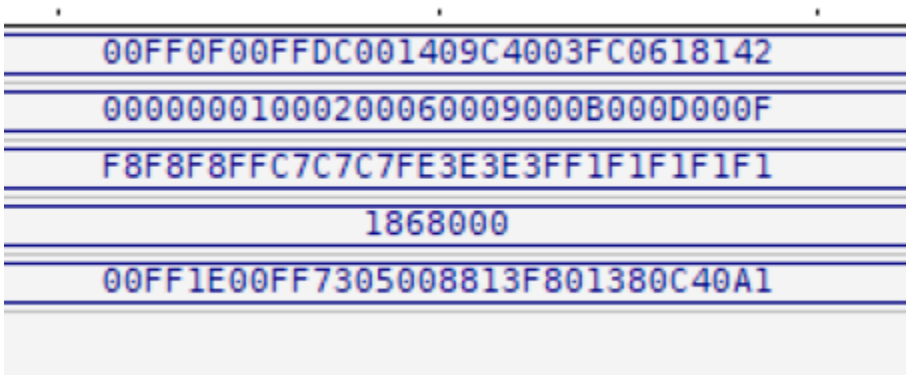
OR:



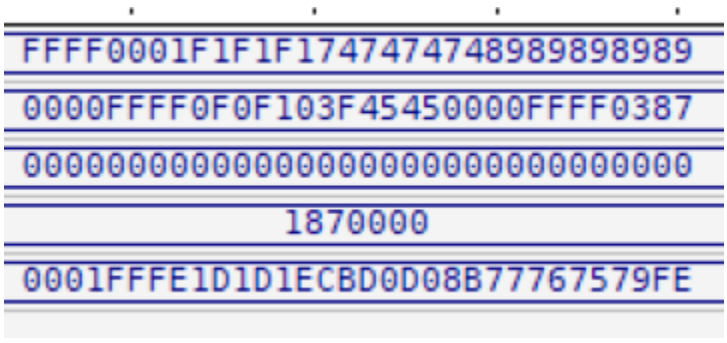
CLZH:



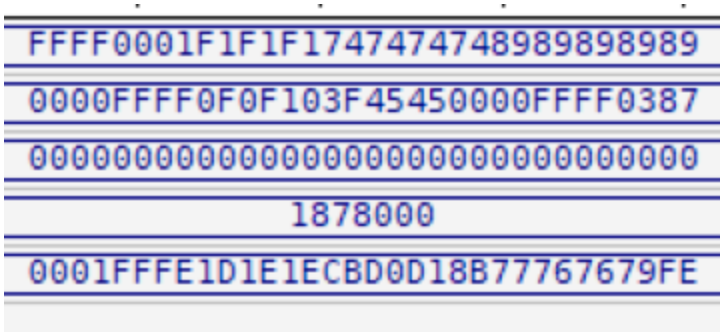
RLH:



SFWU:

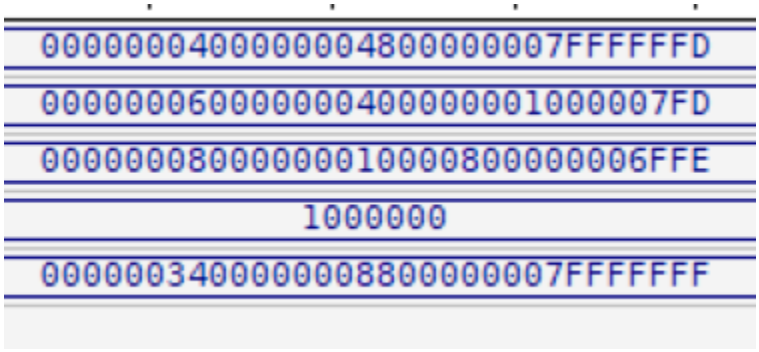


SFHS:

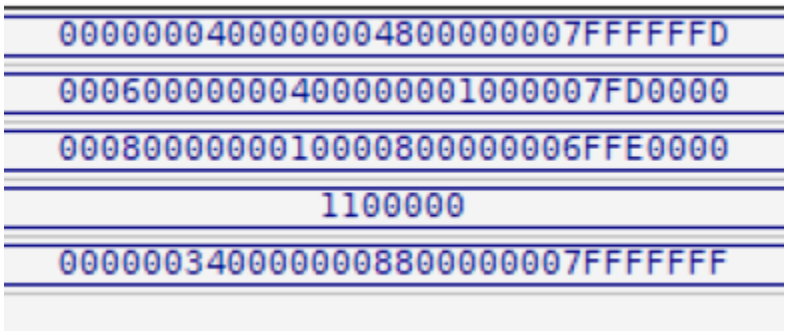


4.2 Multiply-Add and Multiply-Subtract R4-Instruction Format

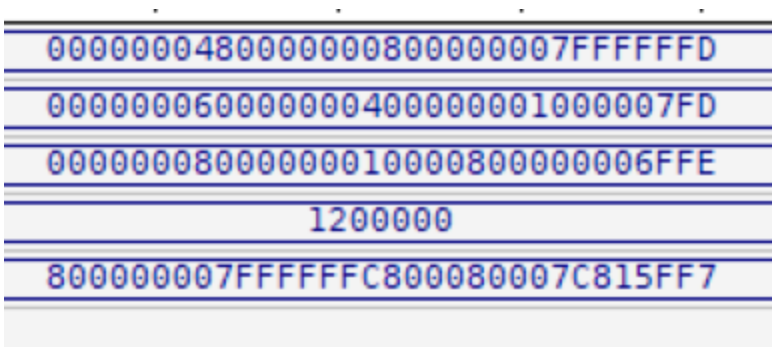
Signed Integer Multiply-Add Low with Saturation:



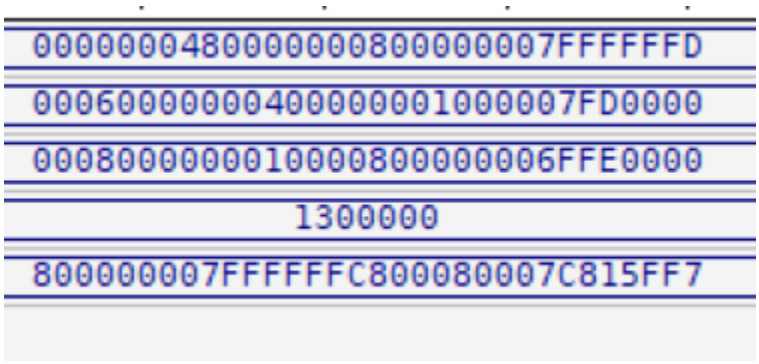
Signed Integer Multiply-Add High with Saturation:



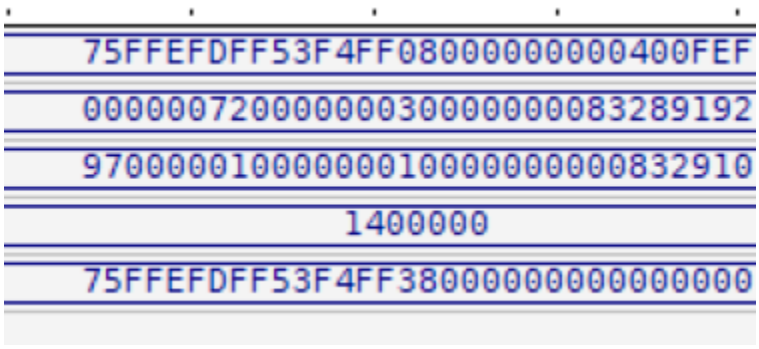
Signed Integer Multiply-Subtract Low with Saturation:



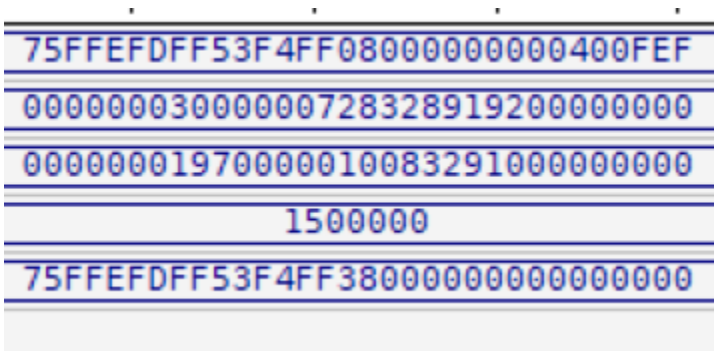
Signed Integer Multiply-Subtract High with Saturation:



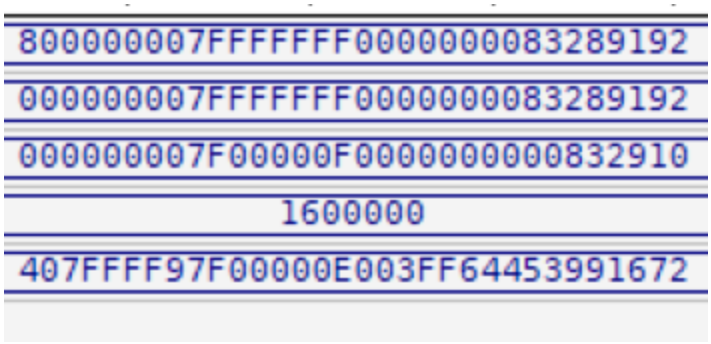
Signed Long Multiply-Add Low with Saturation:



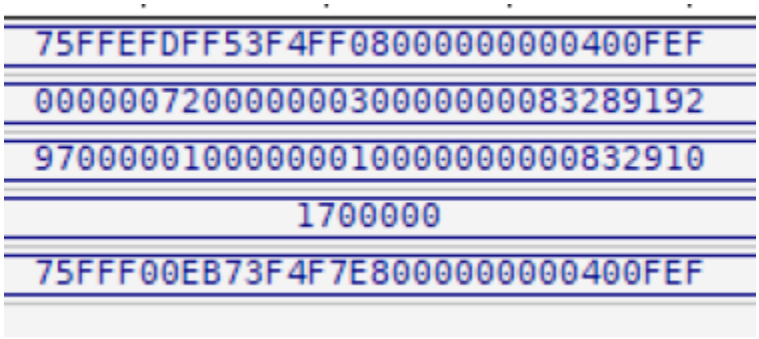
Signed Long Multiply-Add High with Saturation:



Signed Long Multiply-Subtract Low with Saturation:



Signed Long Multiply-Subtract High with Saturation:



Conclusions/Final statements :

In conclusion, our multimedia processing unit effectively processes instructions, adhering to the principles of pipelining and using techniques such as forwarding when facing hazards. Utilizing VHDL allowed us to implement complex logic for handling various operational scenarios.

While most instructions were executed efficiently, some required additional verification to ensure accuracy and performance optimization. This project not only reinforced our understanding of complex architecture concepts but also provided practical experience in troubleshooting and enhancing processor functionality with VHDL.

Through this course and final project, we have gained significant insights into the operational challenges and considerations involved in designing and implementing a pipelined system, laying a solid foundation for future explorations in computer architecture