AMERICAN UNIVERSITY OF BEIRUT
MAROUN SEMAAN FACULTY OF ENGINEERING & ARCHITECTURE

American University of Beirut

Maroun Semaan Faculty of Engineering and Architecture

Department of Electrical and Computer Engineering

EECE 321: Computer Organization

Datapath Design Project

Floating Point Unit

Professor: Mazen Saghir

Team Members: Serena Stephan, Tony Abi Haidar, Joud Senan

Date: 27/4/2025

1. *The Tool and Prompts Used to Implement, Refine, and Test the FPU:*

For the Floating-Point Unit (FPU) implementation, we used Xilinx Vivado 2023.1. The editing tool used is Vivado built-in text editor, Vivado Synthesis for synthesis, and Vivado XSim Simulator for simulation.

Description of our Approach:

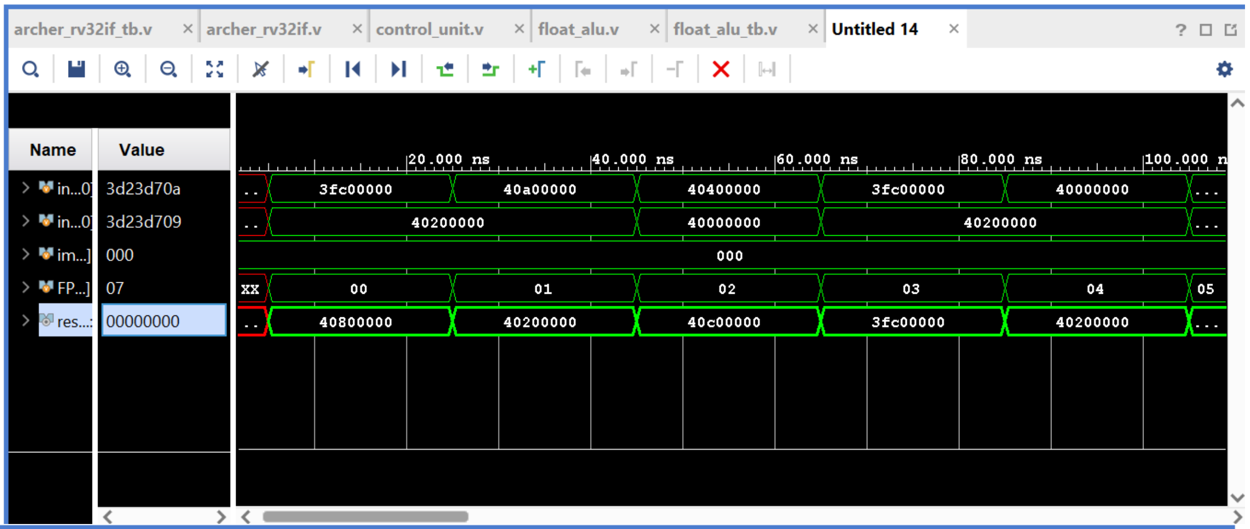
- We modularized the FPU into blocks: fpu.v containing add_fp, sub_fp, mul_fp, max_fp, min_fp, le_fp, lt_fp, and eq_fp.
- We refined floating-point addition and multiplication to handle corner cases: exact zeros, NaNs, ±∞, and normalized/subnormal numbers.
- For testing the fpu code, we got a Verilog Testbench generated by ChatGPT (fpu_tb.v) providing hexadecimal IEEE-754 inputs and verified outputs against expected values.
- We revised all the other codes to check for consistency including: mux2to1, sram, lmb, branch_cmp, alu, immgen, rom, add4, and pc.
- We designed a dedicated Control Unit (control_unit.v) to generate all necessary control signals for both integer and floating-point instructions. It dynamically detects floating-point operations based on the funct7 and funct3 fields and issues the appropriate FPUOp, FPStart, FPRegWrite, and FPToReg signals alongside classical RV32I control signals.
- We implemented two register files (regfile.v and float_regfile.v) to separately manage general-purpose integer registers and floating-point registers. The integer register file ensures x0 remains hardwired to zero, while the floating-point register file handles independent floating-point storage without enforcing f0 = 0, as per RISC-V convention and as instructed in this project.
- During simulation, the control unit properly selects whether operations were integer-based or floating-point-based, and the datapath dynamically routes reads/writes to either the integer or float register file accordingly.
- We tested the fpu logic using a standalone testbench (mentioned above).

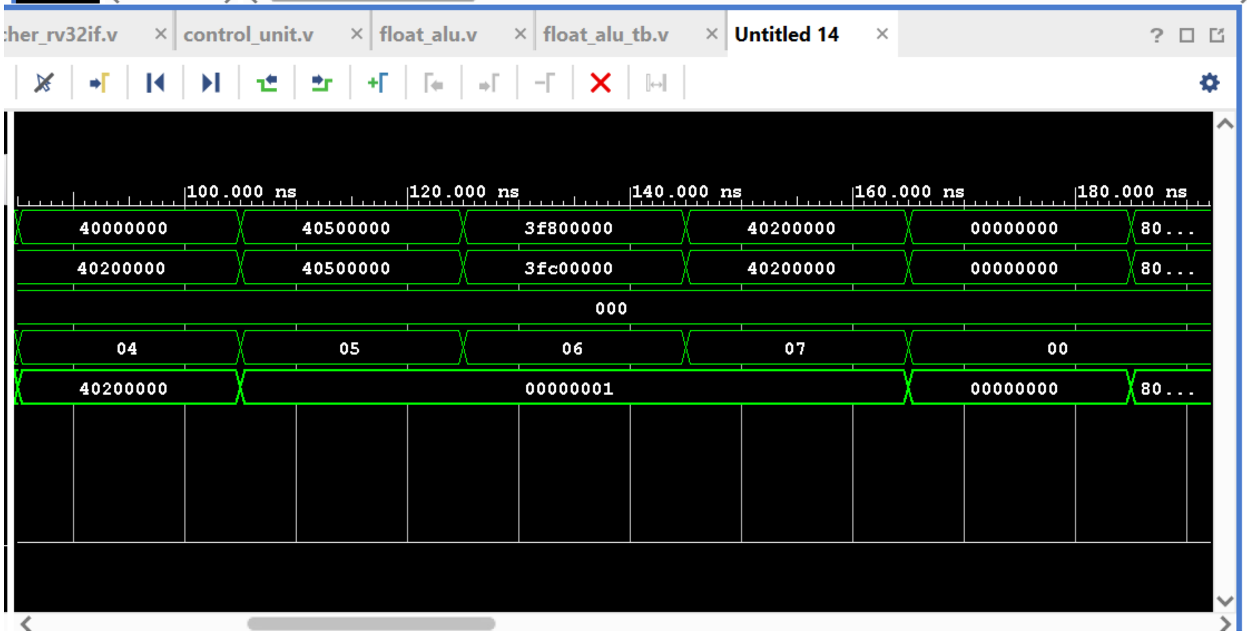
- **Test Cases:**

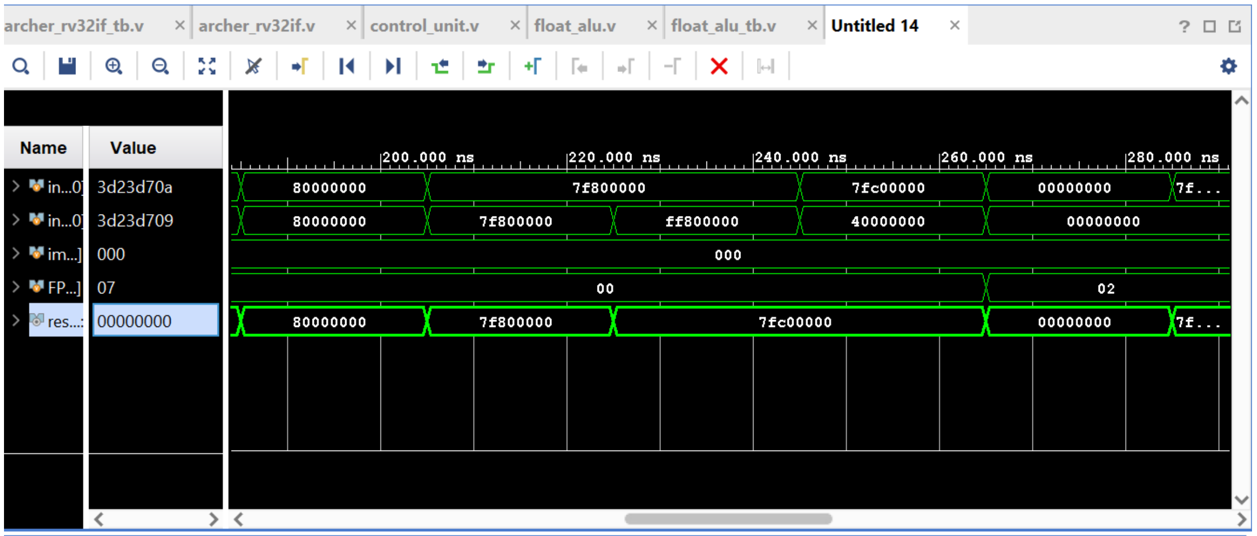
Operation	Inputs	Output
Addition	1.1 + 1.15	40800000
Subtraction	5.0 - 2.5	40200000
Multiplication	3.0 * 2.0	40c00000
Min	(1.5, 2.5)	3fc00000
Max	(2.0, 2.5)	40200000
Equality Check	3.25 == 3.25	1
Less Than	1.0 < 1.5	1
Less Than or Equal	2.5 <= 2.5	1
Special Case	+0.0 + +0.0	00000000
Special Case	-0.0 + -0.0	80000000
Special Case	+Inf + +Inf	7f800000
Special Case	+Inf + -Inf	7fc00000
Special Case	NaN + 2.0	7fc00000

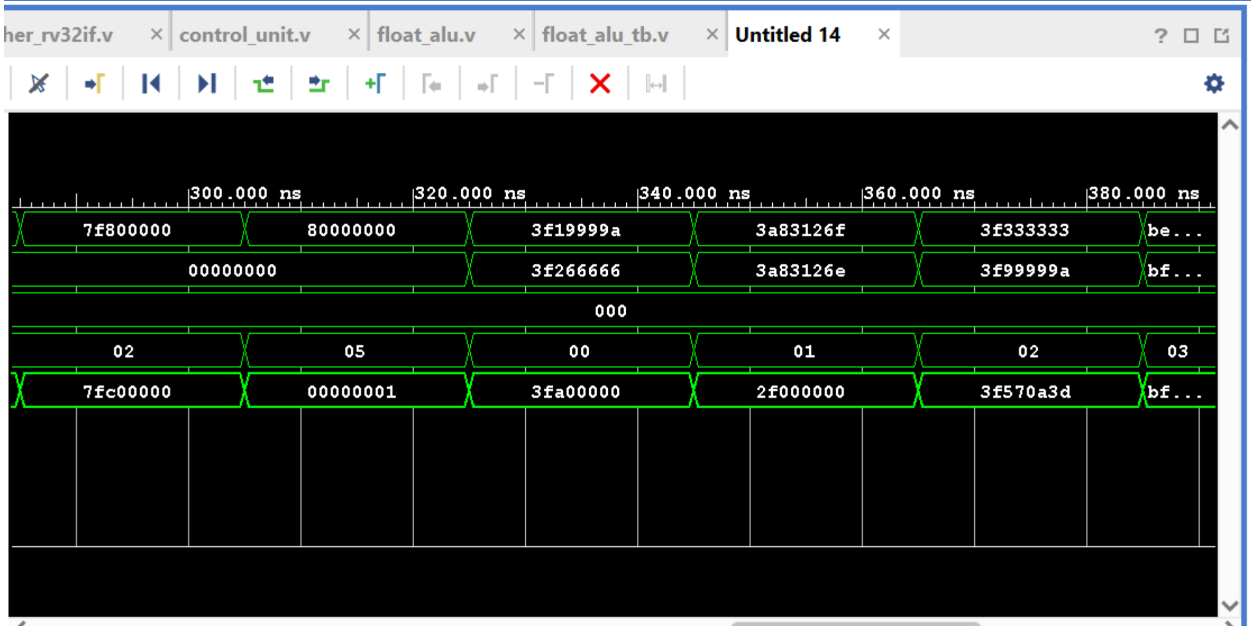
Special Case	0 * 0	00000000
Special Case	Inf * 0	7fc00000
Special Case	+0.0 == -0.0	1
Add Small Fractions	1.1 + 1.15	3fa00000
Subtract tiny numbers		2f000000
MUL tiny fraction	0.7 * 1.2	3f570a3d
MIN negative small	(-0.2, -0.5)	bf000000
MAX tiny positive vs negative	(1e-7, -1e-7)	33d6bf95
LE close decimals	0.04 <= 0.0399999	0

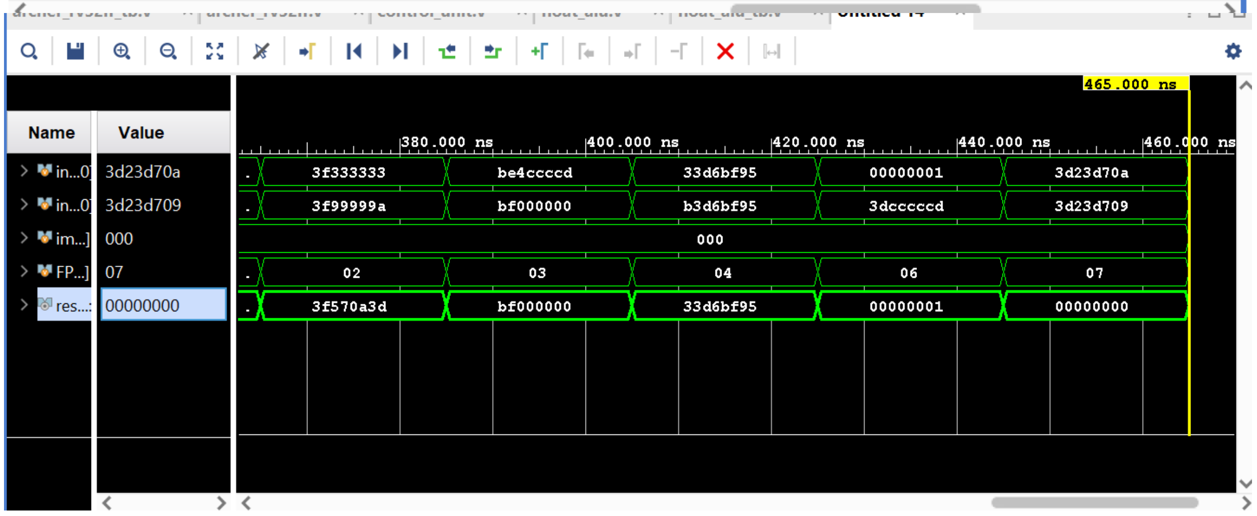
2. ***Vivado simulator waveforms demonstrating successful testing of FPU functionality including operations involving exact zero, ±∞, and NaN:***



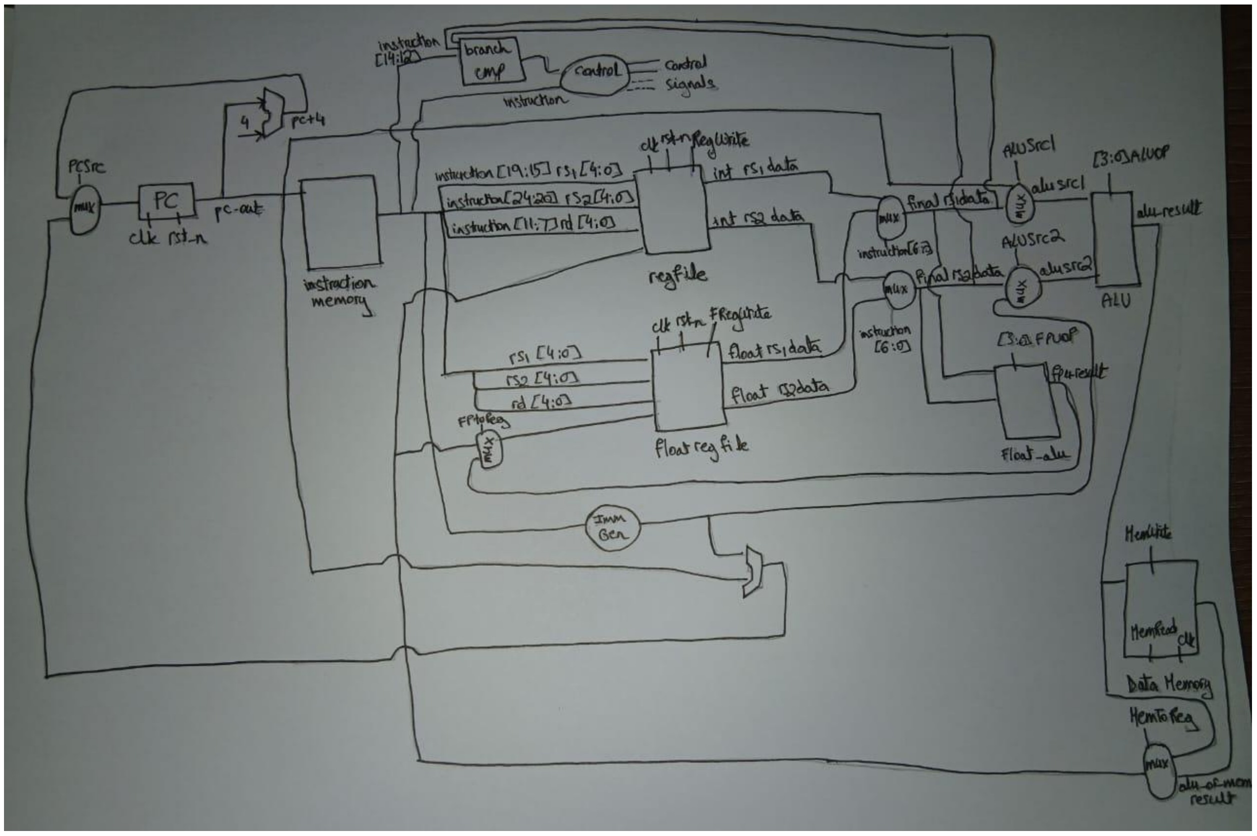



archer_rv32if_tb.v
archer_rv32if.v
control_unit.v
float_alu.v
float_alu_tb.v
Untitled 14
Name
Value
in...0] 3d23d70a
in...0] 3d23d709
im...] 000
FP...] 07
res...: 00000000
200.000 ns
220.000 ns
240.000 ns
260.000 ns
280.000 ns
80000000 7f800000 7fc00000 00000000 7f...
80000000 7f800000 ff800000 40000000 00000000
000
00 02
80000000 7f800000 7fc00000 00000000 7f...


her_rv32if.v
control_unit.v
float_alu.v
float_alu_tb.v
Untitled 14
300.000 ns
320.000 ns
340.000 ns
360.000 ns
380.000 ns
7f800000 80000000 3f19999a 3a83126f 3f333333 be...
00000000 3f266666 3a83126e 3f99999a bf...
000
02 05 00 01 02 03
7fc00000 00000001 3fa00000 2f000000 3f570a3d bf...


465.000 ns
Name
Value
in...0] 3d23d70a
in...0] 3d23d709
im...] 000
FP...] 07
res...: 00000000
380.000 ns
400.000 ns
420.000 ns
440.000 ns
460.000 ns
3f333333 be4ccccd 33d6bf95 00000001 3d23d70a
3f99999a bf000000 b3d6bf95 3dcccccd 3d23d709
000
02 03 04 06 07
3f570a3d bf000000 33d6bf95 00000001 00000000

3. *Detailed Datapath Block Diagram:*



4. *Short Description of Input Ports, Output Ports and Block Functionality for New/Modified Datapath Blocks:*

- ***Control Unit:***

  Input Ports:
  - instruction (32 bits): Fetched instruction word containing opcode, funct3, funct7, rs1, rs2, rd…
  - BranchCond (1 bit): Result of branch comparison (1 = branch taken, 0 = branch not taken).

  Output Ports:
  - Jump (1 bit): High when instruction is jal or jalr, indicating an unconditional jump.
  - Lui (1 bit): High is the instruction is lui.
  - PCSrc (1 bit): Selects program counter source: (branch target / jump address) or PC+4 (sequential).
  - RegWrite (1 bit): Enables write operation in the integer register file.
  - ALUSrc1 (1 bit): Selects first ALU operand: PC (ALUSrc1=1) or register data (ALUSrc1=0).

- ALUSrc2 (1 bit): Selects second ALU operand: immediate (ALUSrc2=1) or register data (ALUSrc2=0).
- ALUOp (4 bits): The opcode which specifies which ALU operation to perform.
- MemWrite (1 bit): Enables memory write.
- MemRead (1 bit): Enables memory read.
- MemToReg (1 bit): Selects memory output as data to write back to register file.
- FPRegWrite (1 bit): Enables write operation in the floating-point register file.
- FPToReg (1 bit): Selects floating-point load result for writing back into the register file.
- FPUOp (4 bits): The opcode which specifies which floating point operation to perform.
- FPStart (1 bit): Start signal for floating-point operations.

Block Functionality:

The Control Unit decodes the 32-bit instruction input based on the opcode field and generates all necessary control signals for the datapath. It uses BranchCond signal for branching instructions and sets PCSrc accordingly. It handles arithmetic and logic instructions, memory instructions, jump instructions, and floating-point instructions.

- ***Floating-Point Unit:***

Input Ports:
inputA (32 bits): First input operand in IEEE-754 single-precision floating-point format.
inputB (32 bits): Second input operand in IEEE-754 single-precision floating-point format.
imm (12 bits): Immediate field.
FPU_OP (5 bits): Control signal selecting the floating-point operation.

Output Ports:
result (32 bits): Computed result of the selected floating-point operation in IEEE-754 format.

Block Functionality:

The FPU performs single-precision (32-bit) floating-point operations based on IEEE-754 standard. Operations supported are the following: addition, subtraction, multiplication, minimum, maximum, equal comparison, less than comparison, and less than or equal comparison. The FPU extracts the sign, exponent, and mantissa from both input operands. It can also handle special cases like addition and multiplication involving zero, infinity handling, and NaN for invalid operations.

The output result is formatted back to the IEEE-754 single-precision 32-bit floating-point format.

- ***Top-Level Datapath (archer_rv32if.v):***

  Input Ports:
  - clk (1 bit): System clock signal.
  - rst_n (1 bit): Active-low reset signal to reset the processor components.

  Block Functionality:

  It implements a single cycle RV32IF processor with support for floating-point operations. It fetches instructions from Instruction Memory, the rom, decodes them using the control unit, and processes integer and floating-point operations through the separate ALU and FPU units.

  We included 2 register files: regfile for integer registers and float_regfile for floating-point registers.

  lmb, connected to sram, handles load and store operations.

  branch_cmp and control unit handle branch decisions.

  The program counter value (pc) is managed by branching, jumping, or sequential incrementation (add4).

- ***Floating-Point Register File:***

  Input Ports:
  - clk (1 bit): clock signal for synchronous writes.
  - rst_in (1 bit): active-low reset signal to initialize all floating-point registers.
  - FRegWrite (1 bit): write enable signal for the floating-point register file.
  - rs1 (5 bits): source register 1 address.
  - rs2 (5 bits): source register 2 address.
  - rd (5 bits): destination register address.
  - datain (32 bits): data to be written into the register file at address rd.

  Output Ports:
  - regA (32 bits): data read from register at address rs1.
  - regB (32 bits): data read from register at address rs2.

  Block Functionality:

It implements a 32-entry floating-point register file. When rst_n is de-asserted, all floating-point registers are cleared to 0. There is no special handling to prevent writing f0 unlike x0 in regfile which is for integers.

Note: We added to archerdefs the definitions that we needed in these new files accounting for floating-point operations.

5. *Description of the assembly code used to test the floating-point instructions and the corresponding Vivado simulator waveforms demonstrating successful tests:*

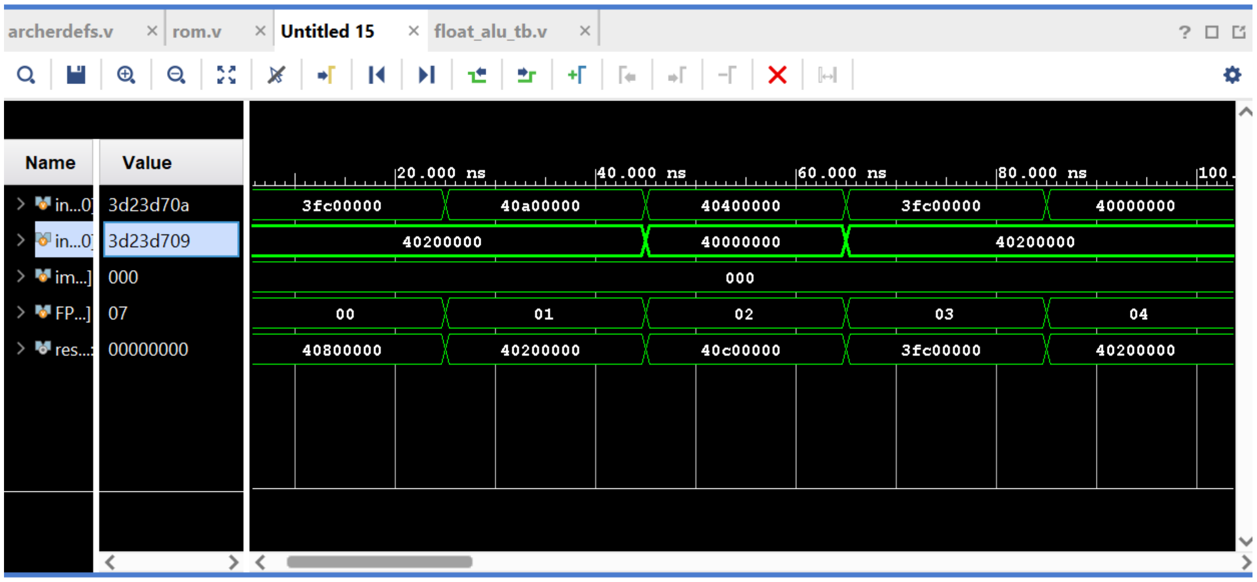
- We also assembled RV32I assembly code (RV32IFtest.s) and loaded it into the ROM, executing floating-point instructions at runtime to verify FPU behavior inside the system. Here are the details:

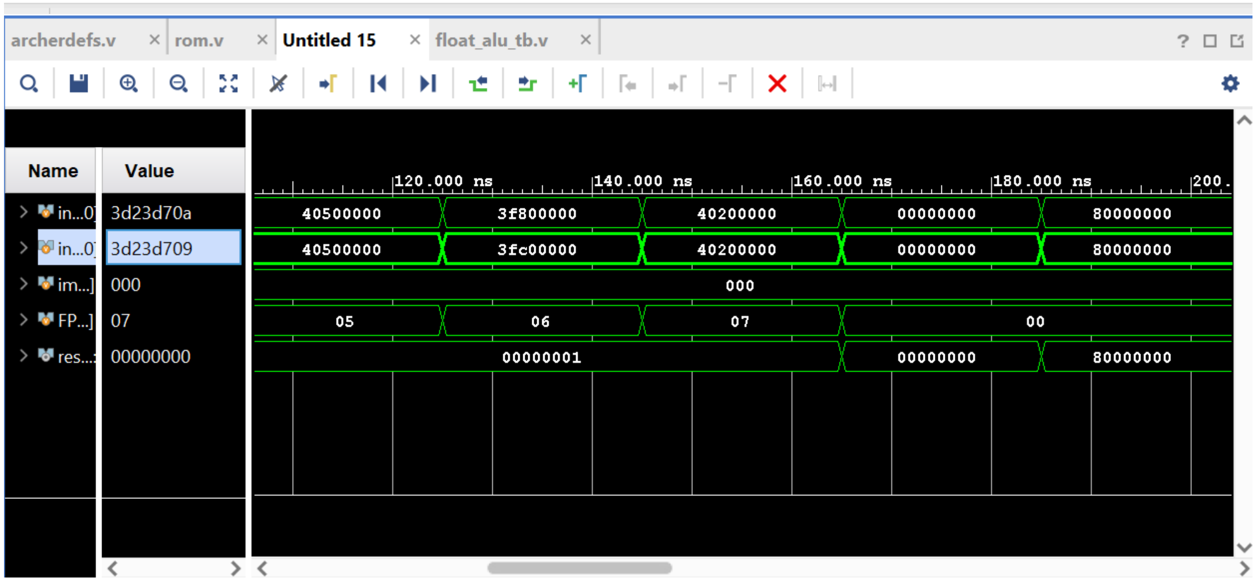
  Floating-Point Test Program:
  A test program was developed to validate the floating-point instructions in the expanded RV32IF datapath. It includes:

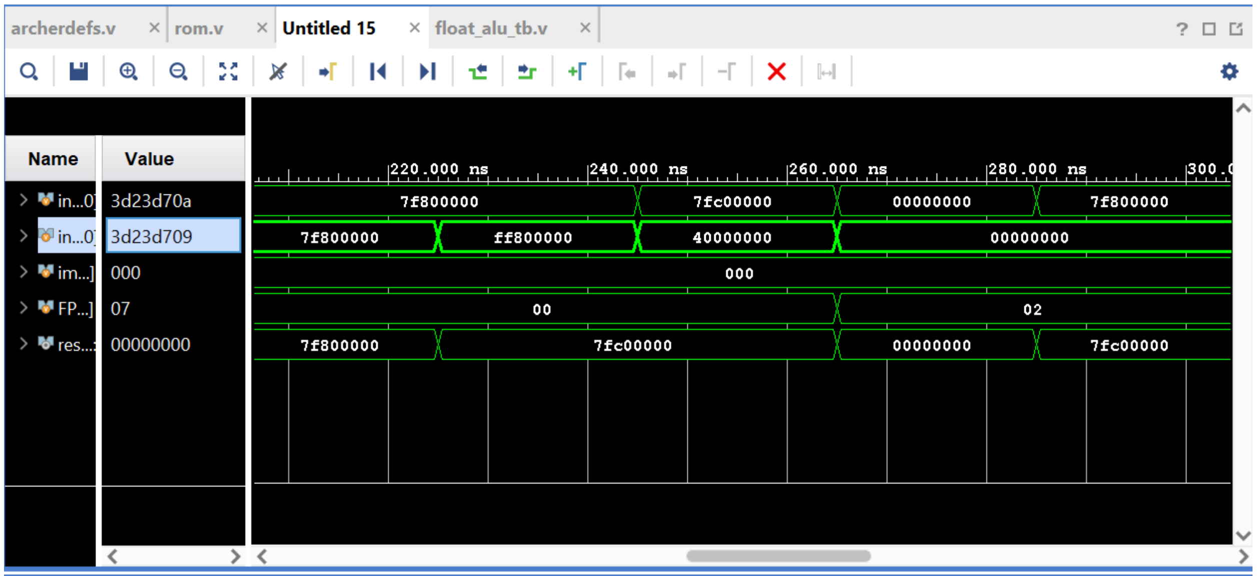
  - **RV32IFtest.s:** RISC-V assembly code that tests floating-point operations across different cases, including zeros, NaNs, and infinities.
  - **RV32IFtest.o:** Machine code generated from the assembly file using Venus Simulator.
  - **RV32IF_test_rom.v:** A Verilog ROM model that stores the machine code for processor simulation.

  Tools Used:
  - Venus Simulator
  - Vivado software

Waveforms:

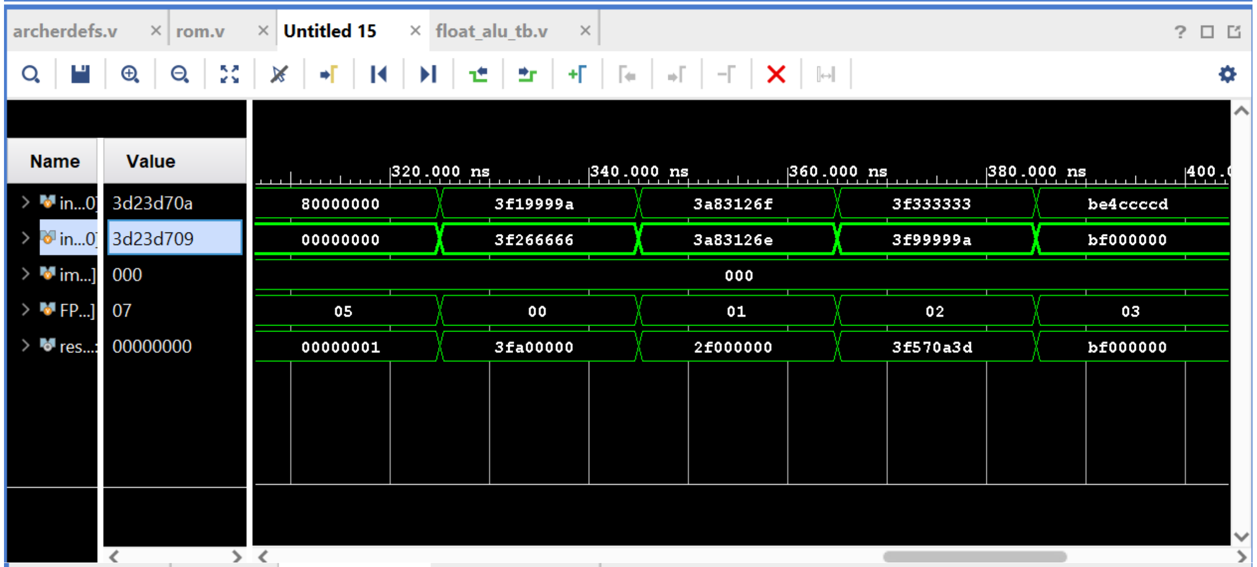

archerdefs.v
rom.v
Untitled 15
float_alu_tb.v
Name
Value
in...0] 3d23d70a
in...0] 3d23d709
im...] 000
FP...] 07
res...: 00000000
20.000 ns
40.000 ns
60.000 ns
80.000 ns
100
3fc00000 40a00000 40400000 3fc00000 40000000
40200000 40000000 40200000
000
00 01 02 03 04
40800000 40200000 40c00000 3fc00000 40200000


archerdefs.v
rom.v
Untitled 15
float_alu_tb.v
Name
Value
in...0] 3d23d70a
in...0] 3d23d709
im...] 000
FP...] 07
res...: 00000000
120.000 ns
140.000 ns
160.000 ns
180.000 ns
200
40500000 3f800000 40200000 00000000 80000000
40500000 3fc00000 40200000 00000000 80000000
000
05 06 07 00
00000001 00000000 80000000



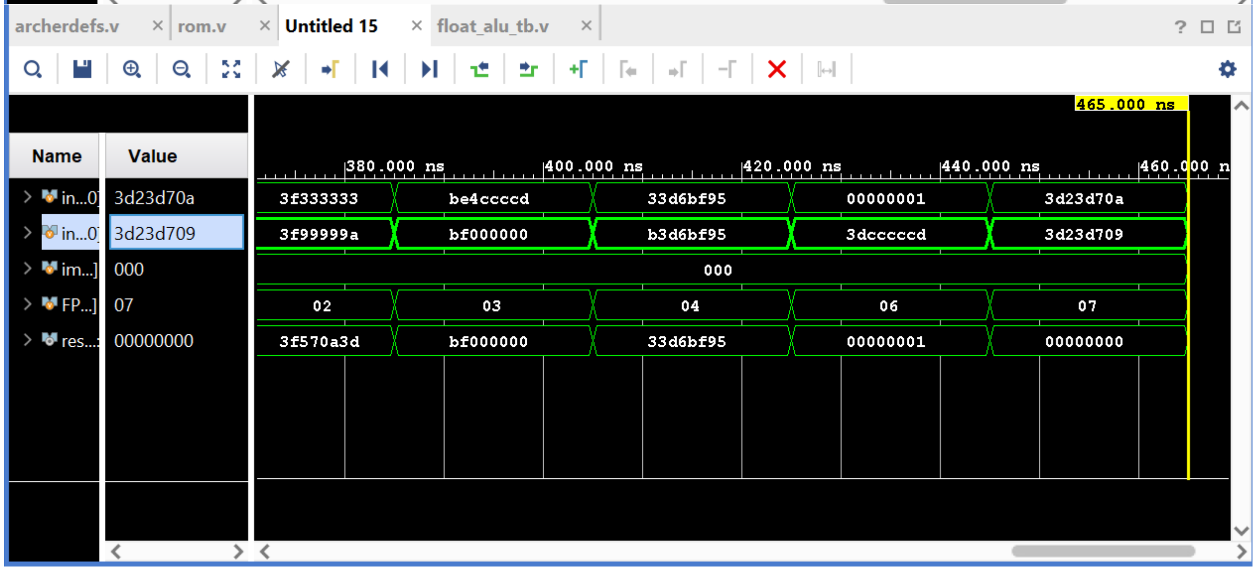
```
archerdefs.v  rom.v  Untitled 15  float_alu_tb.v

Name     Value       220.000 ns      240.000 ns      260.000 ns      280.000 ns      300.0
in...0]  3d23d70a    7f800000                        7fc00000        00000000        7f800000
in...0]  3d23d709    7f800000        ff800000        40000000        00000000
im...]   000         000
FP...]   07          00                                              02
res...]  00000000    7f800000        7fc00000                        00000000        7fc00000
```



```
archerdefs.v  rom.v  Untitled 15  float_alu_tb.v

Name     Value       320.000 ns      340.000 ns      360.000 ns      380.000 ns      400.0
in...0]  3d23d70a    80000000        3f19999a        3a83126f        3f333333        be4ccccd
in...0]  3d23d709    00000000        3f266666        3a83126e        3f99999a        bf000000
im...]   000         000
FP...]   07          05              00              01              02              03
res...]  00000000    00000001        3fa00000        2f000000        3f570a3d        bf000000
```

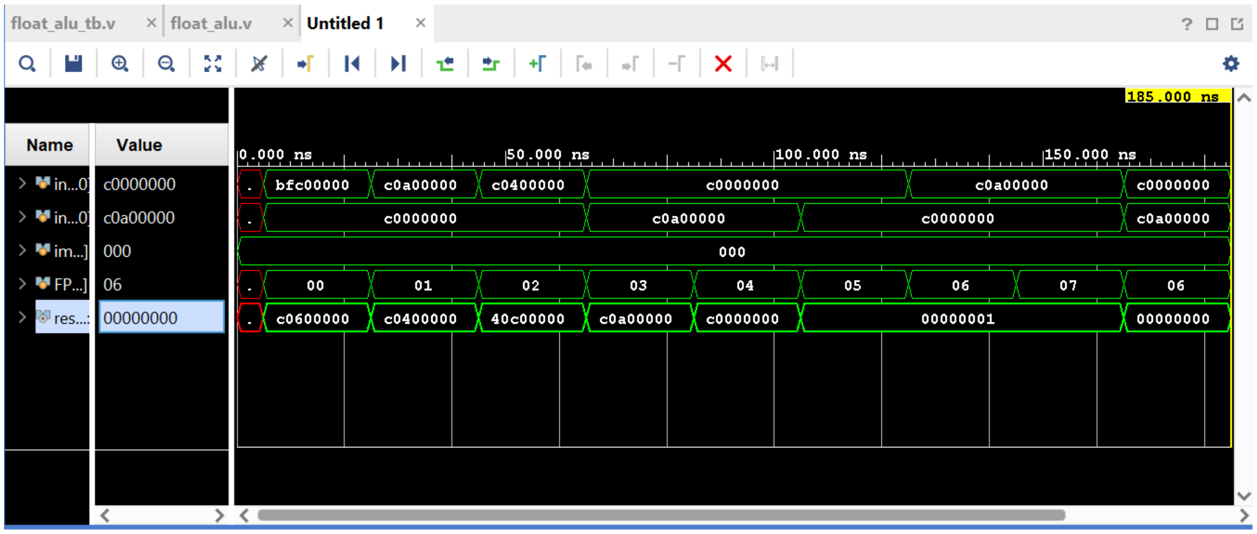


```
archerdefs.v  rom.v  Untitled 15  float_alu_tb.v

                                                                           465.000 ns
Name     Value       380.000 ns      400.000 ns      420.000 ns      440.000 ns      460.000 n
in...0]  3d23d70a    3f333333        be4ccccd        33d6bf95        00000001        3d23d70a
in...0]  3d23d709    3f99999a        bf000000        b3d6bf95        3dcccccd        3d23d709
im...]   000         000
FP...]   07          02              03              04              06              07
res...]  00000000    3f570a3d        bf000000        33d6bf95        00000001        00000000
```

One more thing:

We checked if our fpu can handle operations on negative numbers in a separate test bench, and it worked perfectly! Below is a screenshot of the correctness:



```
in...0]  c0000000   . | bfc00000 | c0a00000 | c0400000 | c0000000 | c0a00000 | c0000000
in...0]  c0a00000   . | c0000000 | c0a00000 | c0000000 | c0a00000
im...]   000        000
FP...]   06         . | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 06
res...:  00000000   . | c0600000 | c0400000 | 40c00000 | c0a00000 | c0000000 | 00000001 | 00000000
```

6. *Contributions of Each Team Member:*

Joud Senan:

- Wrote control_unit.v and archer_rv32if.v
- Wrote the details regarding the implementation and content of the codes in the report.

Serena Stephan:

- Wrote the code of "float_alu"
- Sketched a diagram of the datapath

Tony Abi Haidar:

- Test program (all three versions)
- Modified: archerdefs.v, float_regfile.v

Group contributions:

- Testbenches
- Integrating files
- Debugging