**Design and Implementation of Single Precision FPU in RISC-V Processor**



Final year project report submitted in partial fulfilment of requirement for degree of Bachelors of Science in Electrical Engineering

|  |  |
| --- | --- |
| Arslan Ahmed | NIM-BSEE-2021-09 |
| Nasir Abbas | NIM-BSEE-2021-35 |
|  |  |

Dr. Naureen Shaukat

**Department of Electrical Engineering**

**Namal University, Mianwali**

**2025**

**DECLARATION**

The project report titled “Design and Implementation of Single Precision FPU in RISC-V Processor” is submitted in partial fulfilment of the degree of Bachelors of Science in Electrical Engineering, to the Department of Electrical Engineering at Namal University, Mianwali.

It is declared that this is an original work done by the team members listed below, under the guidance of our supervisor “Dr. Naureen Shaukat”. No part of this project and its report is plagiarised from anywhere, and any help taken from previous work is cited properly.

No part of the work reported here is submitted in fulfilment of requirement for any other degree/ qualification in any institute of learning.

**Team Members Signatures**

Arslan Ahmed NIM-BSEE-2021-09 \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Nasir Abbas NIM-BSEE-2021-35 \_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Supervisor**

Dr. Naureen Shaukat **Signatures with date** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Table of Contents**

[Abstract v](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.f2gstniyite9)

[Introduction 1](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.6n2p1oa7o986)

[Literature Review 2](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.z0w8cjobzefq)

[Methodology 3](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.f3g6fjrimag4)

[3.1 Design Process 3](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.2i1yip35wg8j)

[3.2 Mathematical Model 3](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.q7ihu12sxgw0)

[3.3 Algorithm (if any) 3](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.94ek3lppfi85)

[3.4 Block Diagram 3](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.p32ur6menlf9)

[3.5 Implementation Constraints 3](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.6r9y9235figb)

[Testing 4](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.i2cinuy4exmz)

[Results 5](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.e1iofhqw79yw)

[Discussion 6](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.rf5xmnc9kfin)

[Conclusion 7](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.1szak9z8dhz5)

[Future Work 8](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.dmqny4oqsbw1)

[Reflections on Learning 9](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.rhy0hs3aw715)

[References 10](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.mknwpufhsynt)

[APPENDICES 11](https://docs.google.com/document/d/1JojajA2X1waJxkW96-JP4VHrCy0UUKaF/edit#heading=h.49146o7kl76w)

**List of Figures**

**List of Tables**

**ACKNOWLEDGMENTS**

This project's success was not solely due to our efforts, but also the contributions of many other individuals who played crucial roles in its completion. We would like to express our sincere appreciation to our professors, who have been instrumental in shaping our project's success through their valuable guidance and encouragement.

We are truly appreciative to our Respected supervisor, Dr. Naureen Shaukat, for all of her help, encouragement, and support during this project. Her expertise and views were crucial in forming this work and advancing us. We really appreciate the ways in which she has assisted us, such as with weekly base progress reports, developing complicated concepts, offering constructive feedback, and assisting with obstacles.

We also want to express our sincere gratitude to one another for our outstanding teamwork, commitment, and tenacity during these endeavors. By cooperating, we overcame obstacles and accomplished our goals.

We also express our gratitude to our friends and colleagues for their moral support and useful advice. Finally, we would especially want to thank our families for their unwavering support and tolerance during this difficult project phase.

**Abstract**

RISC-V, an open-source instruction set architecture, has characteristics of high performance, modularity, simplicity and easy expansion therefore numerous open- source cores have been employed in academia and commercial projects within a few years. Floating-Point Units (FPUs) are essential for the calculations of floating point numbers. The project aims to design a Single Precision Floating-Point Unit (FPU) and its integration into RISC-V based processor PicoRV32 which lacks dedicated FPU. FPU is suitable for performing arithmetic and logical operations on floating point numbers. The FPU is designed according to the IEEE-754-2008 standard. To support FPU, different modules like a register file are updated to store floating point numbers, and the decoder is updated to accommodate the floating-point instructions. FPU is integrated as a coprocessor in the PicoRV32 processor core and implemented on Field Programmable Gate Array (FPGA) for hardware verification. The results confirm the successful integration of the designed module in the core. Tests are conducted to compare the core’s performance without FPU and with FPU, ensuring that integrated FPU significantly enhances the core’s performance of handling floating point numbers.

*Chapter 1*

1. **Introduction**

This section covers key preliminary concepts essential for understanding the upcoming project details.

* **RISC-V Processor**
* **Background**
* **PicoRV32**
* **Floating Point Unit**
* **SDG’s Goals**
* **Purpose**

1. **RISC-V Processor**

The processor is the brain of a computer that helps to execute different instructions and perform various calculations. With the advancement in technology, the number of processors is increasing day by day so there are numerous processors available in the market. Companies are working on them to make them according to the needs of the customers and their use. Due to the dynamic nature of the market, the count is not possible. On a regular basis, they are created, modified, and changed to meet better performance and power requirements. RISC-V is a new kind of royalty-free architecture committed by the engineers at University of California, Berkeley. RISC-V is open standard instruction set architecture (ISA) based on the established reduced instruction set principle. Now it is gaining popularity in processor design under open-source licenses in specific applications such as Storage devices, Edge Computing and AI applications.

1. **Background**

RISC-V based processors have changed the way to design processors. In the beginning, the processors were designed using the CISC approach, which was complex for new designers. However, this ISA is open source and allows a wide range of customization options to help computer architects design a processor according to their need [1]. The best thing about RISC-V processors is that they are simple, flexible, open source, and modular. Due its open source, modularity, scalability and simplicity billions of cores have been built by using its ISA. Developers optimize their designs according to their application requirements. This finds applications in both academic and commercial designs because it is free to use.

1. **Purpose**

The main objective of the project is to design a single-precision Floating-Point Unit (FPU) for the PicoRV32 core that will be in accordance with IEEE 754-standard. This FPU will be able to perform accurate floating-point calculations, including many operations such as addition, subtraction, multiplication, and comparisons between numbers. The implementation of FPU integrated core on a Field Programmable Gate Array (FPGA) will enable core testing and validation under real-world conditions, ensuring its reliability and efficiency. 1 Ultimately, this research aims to overcome the constraints of lightweight RISC-V cores through the design of an efficient and resource-conscious single-precision FPU. The project’s success will be demonstrated through rigorous hardware testing and its potential applications, offering valuable insights into academic and technological development.

1. **PicoRV32**

PicoRV32 is a RISC-V core developed by Clifford Wolf[1] and open-sourced for open contributions. PicoRV32 is a CPU core that implements the RISC-V RV32IMC Instruction Set, Integer (I), Multiply/Divide (M) and Compressed (C) extensions. It can be configured as RV32E, RV32I, RV32IC, RV32IM, or RV32IMC core, and optionally contains a built-in interrupt controller. It supports the optional Pico Co-Processor Interface (PCPI) that can be used to implement non-branching instructions in an external coprocessor. By implementations of PCPI any Standard Extension instructions can be integrated in PicoRV32.

1. **Floating Point Unit**

Modern computing systems depend significantly on floating-point arithmetic to process real numbers accurately and with a wide dynamic range. This arithmetic capability is vital for numerous applications, such as scientific studies, digital signal processing and engineering simulations. The IEEE 754 standard for floating-point arithmetic provides accurate processing of floating-point integers [1]. RISC-V architecture has different core processors, the processor cores like PicoRV32, lack a dedicated Floating-Point Unit (FPU), which restricts their performance in high precision tasks. The small cores like the PicoRV32 are designed to minimize hardware complexity and energy consumption, which can compromise their computational strength. But a problem with this is they lack FPUs. This problem can be resolved by introducing a single-precision FPU that adheres with IEEE 754-standard, which will significantly improve PicoRV32’s ability to meet advanced computational requirements. In the integration process, there are some fundamental changes to the PicoRV32, including modifications to its register files, decoder unit and control logic to handle floating-point data and instructions. This will not only improve the core’s computational capabilities but also support the goals of open-source hardware innovation.

1. **SDG’s Goals**

The Sustainable Development Goals (SDGs) are 17 global goals adopted by all United Nations Member States in 2015 as part of the 2030 Agenda for Sustainable Development. These goals aim to address major global challenges such as poverty, inequality, climate change, environmental degradation, peace, and justice. Each goal is broad, measurable, and has specific targets and indicators to track progress.

**SDG Goal 9: Industry, Innovation, and Infrastructure**

Sustainable Development Goal 9 (SDG 09) includes building resilient infrastructure, promoting inclusive and sustainable industrialization, and fostering innovation. Moreover, our work coincides with SDG 09 by fostering innovation. By equipping PicoRV32 with enhanced floating-point features, the project contributes to developments in embedded systems, scientific research, and computationally intensive fields.

**Objectives:**

* Develop quality, reliable, sustainable, and resilient infrastructure
* Promote inclusive and sustainable industrialization
* Enhance scientific research, upgrade technological capabilities, and foster innovation

**Project relation with SDG 09**

Our project involves RISC-V processor enhancement, FPU design, embedded systems, and RISC-V development so it relate to goal 09

1. Innovation & Research
2. Industry Readiness
3. Infrastructure Development

out a brief overview of the project and the background information required to understand different parts of it. Context for the project and reason for its selection should be provided in this section. Use as simplified language as you can (without technical terms and details) so that anyone with some basic knowledge is able to understand.

Mapping of the project goals with the United Nation’s Sustainable Development goals, as well as the effect of the project on environment and sustainability in general should be provided.

It should include the problem statement, context, aims and objectives of the project, and some overview of the solution/ criteria for completion. Provide a summary of the report structure and a brief outline of the chapters to come.

*Chapter 2*

**Literature Review**

The development and optimization of Floating-Point Units (FPUs) have been extensively researched due to their critical role in modern processors. FPUs enable processors to perform floating-point arithmetic efficiently, a fundamental requirement for fields like scientific computation, real-time graphics, and advanced system processing. The IEEE 754 standard, established in 1985, has been instrumental in defining consistent and accurate methods for floating-point representation and operations, ensuring uniformity across hardware and software platforms.

3.1 FPUs in Processor Architectures

 The integration of FPUs into processors significantly enhances their computational capabilities. For instance, Murali Krishna et al. (2013) demonstrated the advantages of incorporating FPUs for handling both simple and complex arithmetic operations [2]. Their design emphasized reducing hardware complexity while improving resource utilization. Similarly, Harshita Nair et al. (2020) presented an efficient implementation of single-precision floating-point operations on FPGA platforms [3]. By employing techniques such as pipelining and truncation, their work achieved notable improvements in speed and area efficiency, adhering to IEEE 754 guidelines. Despite advancements in FPU designs, lightweight cores like PicoRV32 often omit FPUs to maintain simplicity and energy efficiency. This exclusion limits their application in tasks requiring high precision and computational power. Addressing this challenge, Saghir (2010) highlighted the potential of application-specific instruction-set architectures in balancing performance with resource constraints. This perspective aligns with the objectives of this research, which seeks to integrate an efficient FPU into the PicoRV32 core.

 3.2 Challenges in FPU Design

The design of FPUs poses several challenges, including hardware complexity, compliance with IEEE 754 standards, and optimization for power, area, and latency. Innovative architecture is being designed to address these challenges, emphasizing reduced lookup table usage and improved precision without excessive resource overhead. Harshita Nair et al. (2020) further explored trade-offs between latency and area, showcasing designs that achieve compactness and efficiency [3]. 2 For the PicoRV32 core, integrating an FPU requires significant modifications to its microarchitecture. These changes include updating register files to support floating-point data and adapting control logic to handle floating-point instructions. Louca et al. (1996) discussed similar architectural adjustments necessary for enabling floating-point operations in lightweight processors, providing valuable insights for this project.

3.3 Hardware Validation and FPGA Implementation

 The practical validation of FPU designs often involves implementation on FPGA platforms. FPGAs provide a flexible and efficient means to test and refine hardware architectures. Harshita Nair et al. (2020) demonstrated the use of Xilinx FPGAs to validate single-precision floating point designs, highlighting the importance of efficient mapping and synthesis techniques [3]. The Xilinx Artix-7 FPGA, selected for this project, offers an optimal balance between resource availability and computational capability. Testing the designed FPU on an FPGA ensures its functionality and performance under real world conditions. Patterson and Hennessy (2020) emphasized the significance of hardware validation in processor design, noting its role in identifying and addressing architectural bottlenecks [5]. By leveraging FPGA testing, this research aims to refine the integrated PicoRV32 core, ensuring it meets its intended objectives of enhanced computational precision and efficiency.

*Chapter 3*

1. **Methodology**

* **Thoroughly study the RISC-V, PicoRv32 core**

PicoRV32 is a CPU core that implements the [RISC-V RV32IMC Instruction Set](http://riscv.org/). It can be configured as RV32E, RV32I, RV32IC, RV32IM, or RV32IMC core, and optionally contains a built-in interrupt controller.

* **Features and Typical Applications**
* Small (750-2000 LUTs in 7-Series Xilinx Architecture)
* High fmax (250-450 MHz on 7-Series Xilinx FPGAs)
* Selectable native memory interface or AXI4-Lite master
* Optional IRQ support (using a simple custom ISA)
* Optional Co-Processor Interface

This CPU is meant to be used as auxiliary processor in FPGA designs and ASICs. Due to its high fmax it can be integrated in most existing designs without crossing clock domains. When operated on a lower frequency, it will have a lot of timing slack and thus can be added to a design without compromising timing closure. Furthermore it is possible to choose between a dual-port and a single-port register file implementation. The former provides better performance while the latter results in a smaller core. The core exists in three variations:

* ****

**picorv32,**

* ****

**picorv32\_axi**

* 

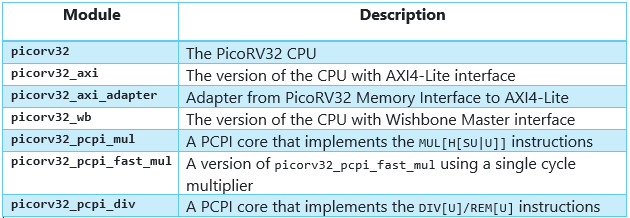
**picorv32\_wb.**

The first provides a simple native memory interface that is easy to use in simple environments. picorv32\_axi provides an AXI-4 Lite Master interface that can easily be integrated with existing systems that are already using the AXI standard. picorv32\_wb provides a Wishbone master interface.

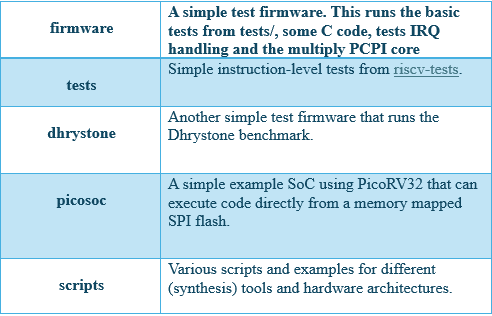
A separate core picorv32\_axi\_adapter is provided to bridge between the native memory interface and AXI4. This core can be used to create custom cores that include one or more PicoRV32 cores together with local RAM, ROM, and memory-mapped peripherals, communicating with each other using the native interface, and communicating with the outside world via AXI4.

The optional IRQ feature can be used to react to events from the outside, implement fault handlers, or catch instructions from a larger ISA and emulate them in software.

The optional Pico Co-Processor Interface (PCPI) can be used to implement non-branching instructions in an external coprocessor. Implementations of PCPI cores that implement the M Standard Extension instructions MUL[H[SU|U]] and DIV[U]/REM[U] are included in this package.



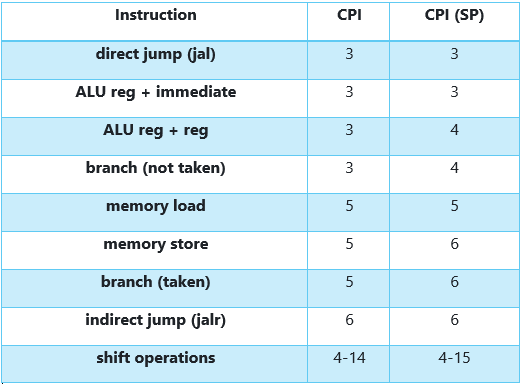
Below are the different test for picorv32



* **Cycles per Instruction Performance**

Unless stated otherwise, the following numbers apply to a PicoRV32 with ENABLE\_REGS\_DUALPORT active and connected to a memory that can accommodate requests within one clock cycle.

The average Cycles per Instruction (CPI) is approximately 4, depending on the mix of instructions in the code. The CPI numbers for the individual instructions can be found in the table below. The column "CPI (SP)" contains the CPI numbers for a core built without ENABLE\_REGS\_DUALPORT.

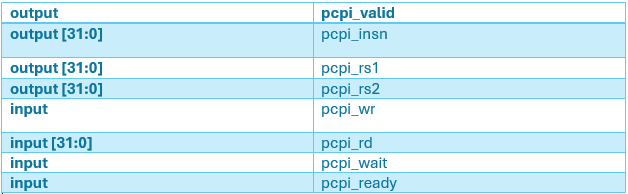


When ENABLE\_MUL is activated, then a MUL instruction will execute in 40 cycles and a MULH[SU|U] instruction will execute in 72 cycles. When ENABLE\_DIV is activated, then a DIV[U]/REM[U] instruction will execute in 40 cycles. When BARREL\_SHIFTER is activated, a shift operation takes as long as any other ALU operation.

The following dhrystone benchmark results are for a core with enabled ENABLE\_FAST\_MUL, ENABLE\_DIV, and BARREL\_SHIFTER options. Dhrystone benchmark results: 0.516 DMIPS/MHz (908 Dhrystones/Second/MHz) For the Dhrystone benchmark the average CPI is 4.100. Without using the look-ahead memory interface , these results drop to 0.305 DMIPS/MHz and 5.232 CPI.

* **Pico Co-Processor Interface (PCPI)**

The Pico Co-Processor Interface (PCPI) can be used to implement non-branching instructions in external cores:



When an unsupported instruction is encountered and the PCPI feature is activated (see ENABLE\_PCPI above), then pcpi\_valid is asserted, the instruction word itself is output on pcpi\_insn, the rs1 and rs2 fields are decoded and the values in those registers are output on pcpi\_rs1 and pcpi\_rs2. An external PCPI core can then decode the instruction, execute it, and assert pcpi\_ready when execution of the instruction is finished. Optionally a result value can be written to pcpi\_rd and pcpi\_wr asserted. The PicoRV32 core will then decode the rd field of the instruction and write the value from pcpi\_rd to the respective register. When no external PCPI core acknowledges the instruction within 16 clock cycles, then an illegal instruction exception is raised and the respective interrupt handler is called. A PCPI core that needs more than a couple of cycles to execute an instruction, should assert pcpi\_wait as soon as the instruction has been decoded successfully and keep it asserted until it asserts pcpi\_ready. This will prevent the PicoRV32 core from raising an illegal instruction exception.

* **Building a pure RV32I Toolchain**

Run the following commands to build the complete toolchain:

* **make download-tools**
* **make -j$(nproc) build-tools**

The default settings in the [riscv-tools](https://github.com/riscv/riscv-tools) build scripts will build a compiler, assembler and linker that can target any RISC-V ISA, but the libraries are built for RV32G and RV64G targets.

Follow the instructions below to build a complete toolchain (including libraries) that target a pure RV32I CPU.

Tjhe following commands will build the RISC-V GNU toolchain and libraries for a pure RV32I target, and install it in /opt/riscv32i:

# Ubuntu packages needed:

sudo apt-get install autoconf automake autotools-dev curl libmpc-dev \

    libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo \

    gperf libtool patchutils bc zlib1g-dev git libexpat1-dev

sudo mkdir /opt/riscv32i

sudo chown $USER /opt/riscv32i

Git clone https://github.com/riscv/riscv-gnu-toolchain riscv-gnu-toolchain-rv32i

cd riscv-gnu-toolchain-rv32i

git checkout 411d134

git submodule update --init --recursive

mkdir build; cd build

../configure --with-arch=rv32i --prefix=/opt/riscv32i

make -j$(nproc)

The commands will all be named using the prefix riscv32-unknown-elf-, which makes it easy to install them side-by-side with the regular riscv-tools (those are using the name prefix riscv64-unknown-elf- by default).

Alternatively you can simply use one of the following make targets from PicoRV32's Makefile to build a RV32I[M][C] toolchain. You still need to install all prerequisites, as described above. Then run any of the following commands in the PicoRV32 source directory:

|  |  |  |
| --- | --- | --- |
| **Command** | **Install Directory** | **ISA** |
| make -j$(nproc) build-riscv32i-tools | /opt/riscv32i/ | RV32I |
| make -j$(nproc) build-riscv32ic-tools | /opt/riscv32ic/ | RV32IC |
| make -j$(nproc) build-riscv32im-tools | /opt/riscv32im/ | RV32IM |
| make -j$(nproc) build-riscv32imc-tools | /opt/riscv32imc/ | RV32IMC |

Or simply run make -j$(nproc) build-tools to build and install all four tool chains.

By default calling any of those make targets will (re-)download the toolchain sources. Run make download-tools to download the sources to /var/cache/distfiles/ once in advance.

*Note: These instructions are for git rev 411d134 (2018-02-14) of riscv-gnu-toolchain.*

**Linking binaries with newlib for PicoRV32**

The tool chains (see last section for install instructions) come with a version of the newlib C standard library.

Use the linker script [firmware/riscv.ld](https://github.com/YosysHQ/picorv32/blob/main/firmware/riscv.ld) for linking binaries against the newlib library. Using this linker script will create a binary that has its entry point at 0x10000. (The default linker script does not have a static entry point, thus a proper ELF loader would be needed that can determine the entry point at runtime while loading the program.)

Newlib comes with a few syscall stubs. You need to provide your own implementation of those syscalls and link your program with this implementation, overwriting the default stubs from newlib. See syscalls.c in [scripts/cxxdemo/](https://github.com/YosysHQ/picorv32/blob/main/scripts/cxxdemo) for an example of how to do that.

It must include the following:

1. **Design Process**

Explain the design process for the project in detail. This must include the design standards used (if any), and the evidence for best design practices followed.

1. **Mathematical Model**

Support the project design with a mathematical model.

1. **Algorithm (if any)**

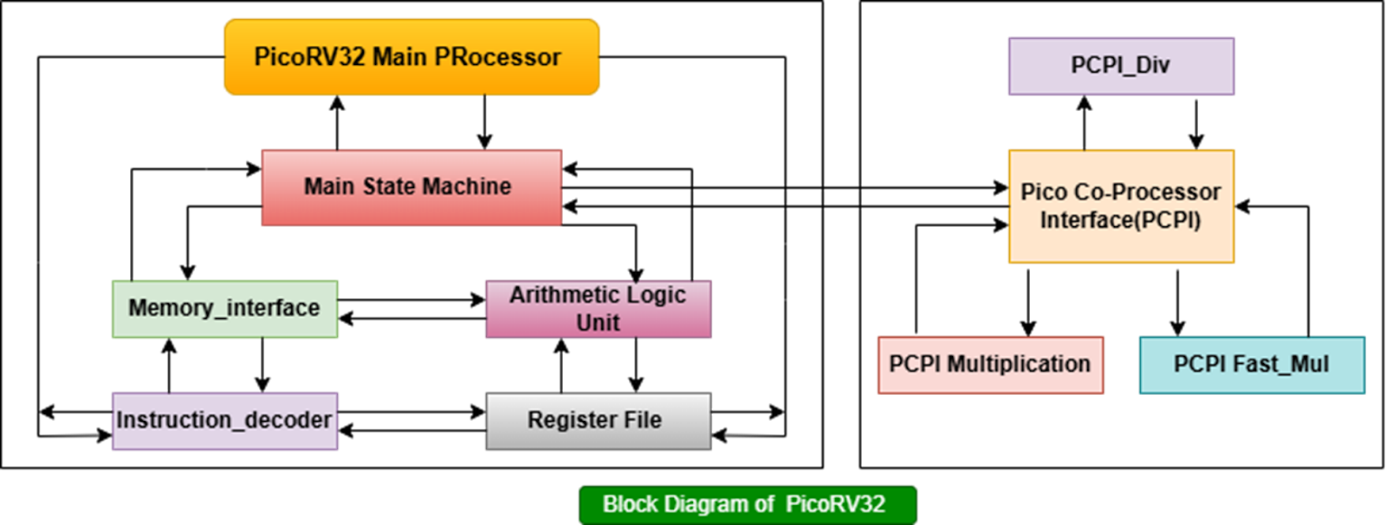
Explain the algorithm for logic etc. used in the project.

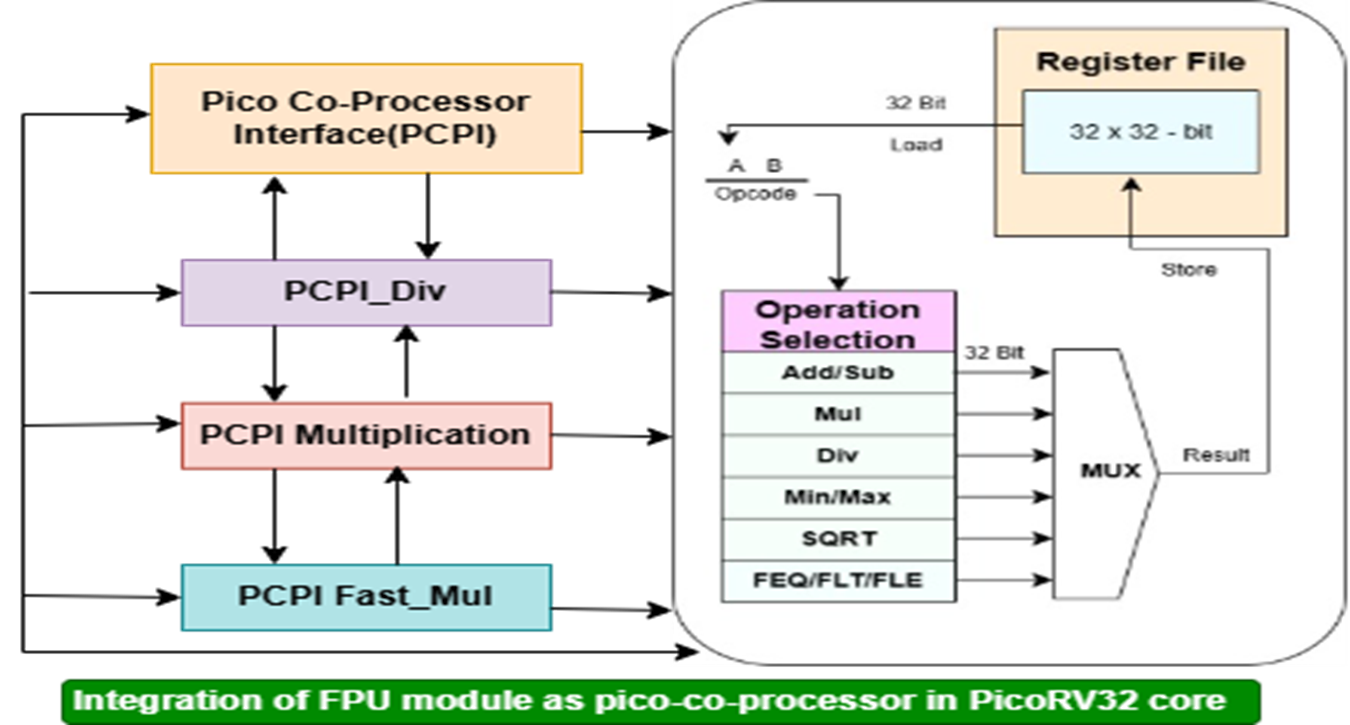
1. **Block Diagram**

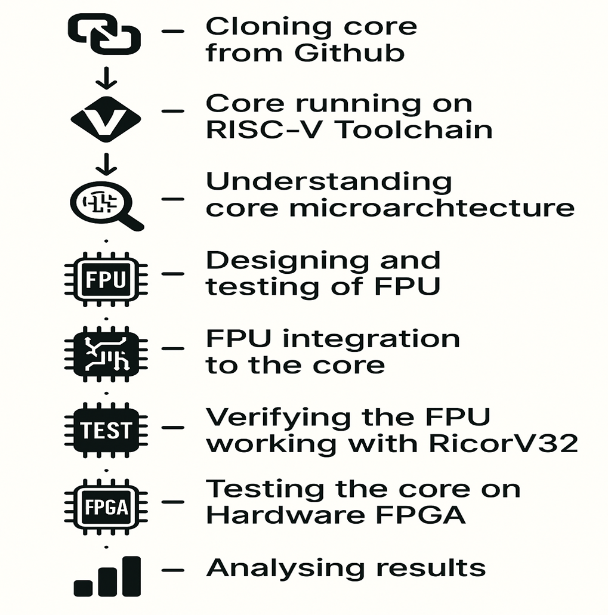
Describe your system with the help of a clear and concise block diagram.

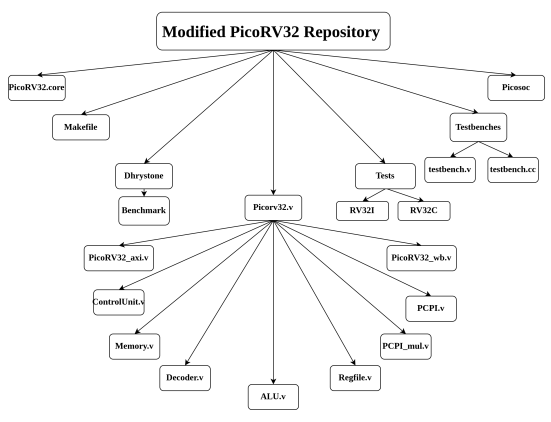
1. **Implementation Constraints**

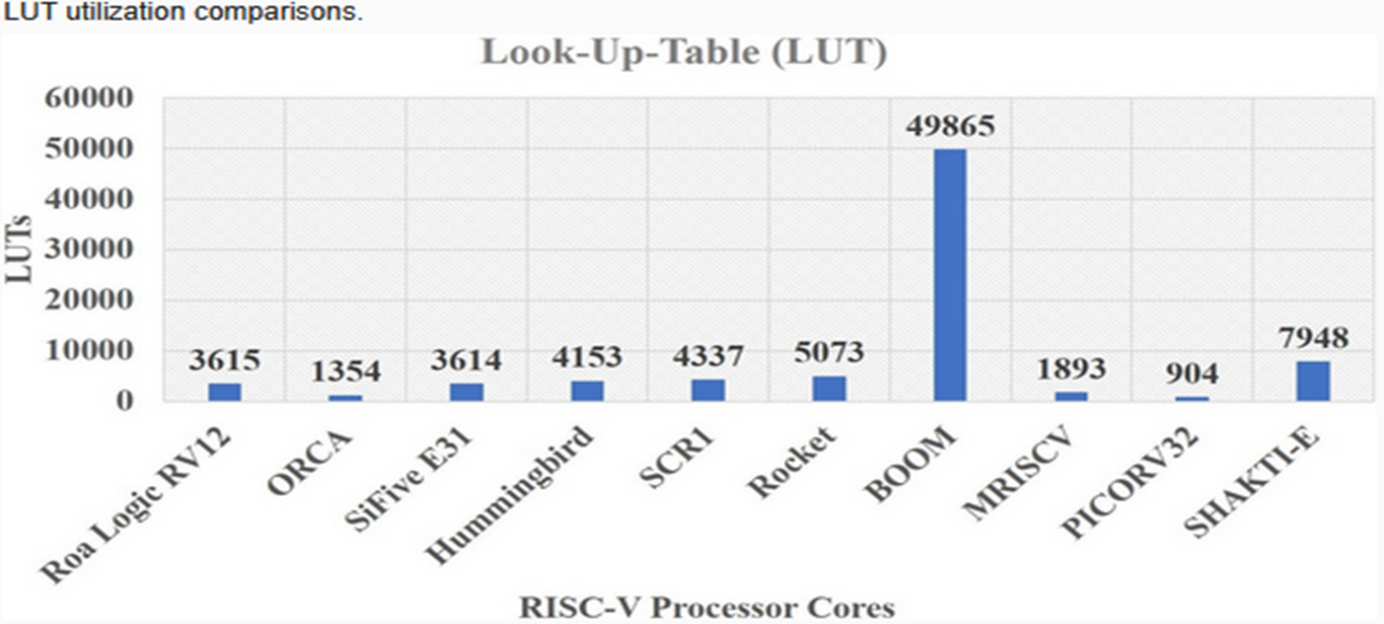
Provide information about the constraints that effected the project implementation, and the measures used as a solution as a result.











*Chapter 4*

**Testing**

Include the various test cases used to check the working of the solution. Explain the testing methodologies used in reference to the core functionalities defined at the proposal stage.

This chapter may be merged into either the methodology or the results if desired, but test cases must be described.

*Chapter 5*

**Results**

Show a complete assessment of the solution implemented through the various test cases described in the previous chapter. The results must be supported by figure, plots, comparison tables etc. to increase understanding of the reader. Results section should not include all test results if there are too many, but the major trends etc. must be clearly noted and should be observable through the supporting evidence.

The complete results (if too many) may be included as an appendix.

*Chapter 6*

**Discussion**

Discuss the results in an objective manner. The discussion section should include a review of what you were trying to accomplish? (the original problem), what actually happened? (results), why does that matter? (implications), how does the project not achieve all the best case scenarios? (limitations), what you were not able to do and why? (failings and their reasons).

Discussion should not include any new methods or results not provided in the methodology or results section.

*Chapter 7*

**Conclusion**

The conclusion should complement the introduction. Provide a brief summary of how the project addresses the problem specified in the introduction. Provide overview of the preceding sections, including the methods used to implement the solution, those used to evaluate its working and how the evaluation results show that the problem has been addressed.

Similar to discussion chapter, the conclusions should not provide any new information, not already stated in the relevant sections of the report.

*Chapter 8*

**Future Work**

Provide an overview of recommendations, and in which direction the project could be taken in the future, giving some information about what resources would be required to further develop it in that direction.

*Chapter 9*

**Reflections on Learning**

Reflect on the overall learning from the project. Clearly communicate the technical and non-technical skills that were developed as a result of the project. Explain how the project has helped in the personality development, and what could have been improved to achieve a better result.

**References**

Provide a comprehensive list of references cited using the IEEE format.

**APPENDICES**

<Attach any extra information (complete code or a link to it, all results etc.), references to any documents as appendices starting from next page>

**Integrating FPU in PicoRV32 as a Coprocessor (PCPI-FPU Integration)**

To integrate the **Floating Point Unit (FPU)** into the **PicoRV32 core** as a **coprocessor module**, we will utilize **PCPI (Pico Co-Processor Interface)**, which is already available in PicoRV32. This method allows the FPU to execute **floating-point instructions** while keeping the main core lightweight.

**Step-by-Step FPU Integration in PicoRV32**

**1️⃣ Understanding PCPI in PicoRV32**

PicoRV32 provides a **PCPI (Pico Co-Processor Interface)**, which allows connecting **external processing units** like **FPU, custom accelerators, or other hardware extensions**.

The PCPI interface consists of the following **control signals**:

* pcpi\_valid: Indicates that an instruction requires coprocessor execution.
* pcpi\_insn: The **instruction** being executed.
* pcpi\_rs1: First operand (from register file).
* pcpi\_rs2: Second operand (from register file).
* pcpi\_wr: Coprocessor writes back the result.
* pcpi\_rd: The **computed result** from the coprocessor.
* pcpi\_ready: Signals that the coprocessor has finished execution.

**2️⃣ Modify PicoRV32 to Recognize Floating-Point Instructions**

To integrate the FPU, the **instruction decoder** in PicoRV32 needs to recognize **floating-point instructions** and send them to the FPU via the **PCPI interface**.

1. **Identify Floating-Point Instructions**
   * In RISC-V, floating-point instructions have the opcode **1010011** (7'b1010011 in binary).
   * The **funct3** field determines the specific operation (ADD, SUB, MUL, DIV, etc.).
2. **Modify the Instruction Decoder in PicoRV32**
   * Locate the section in picorv32\_core.v where pcpi\_valid is assigned.
   * Add a condition to detect **floating-point instructions (7'b1010011)** and set pcpi\_valid = 1.
   * Ensure pcpi\_insn, pcpi\_rs1, and pcpi\_rs2 are correctly assigned.

**3️⃣ Connect the PCPI-FPU Module**

The **PCPI-FPU module (pcpi\_fpu)** acts as a bridge between the **PicoRV32 core and the FPU**.

1. **Instantiate the pcpi\_fpu Module** in picorv32\_core.v  
   Add the following inside the PicoRV32 core:
2. pcpi\_fpu fpu\_unit (
3. .clk(clk),
4. .reset(reset),
5. .pcpi\_valid(pcpi\_valid),
6. .pcpi\_insn(pcpi\_insn),
7. .pcpi\_rs1(pcpi\_rs1),
8. .pcpi\_rs2(pcpi\_rs2),
9. .pcpi\_wr(pcpi\_wr),
10. .pcpi\_rd(pcpi\_rd),
11. .pcpi\_ready(pcpi\_ready)
12. );
13. **Modify the PCPI Logic in PicoRV32**
    * Locate the section where pcpi\_valid is assigned.
    * Add logic to set pcpi\_valid = 1 when a floating-point instruction is detected:
14. assign pcpi\_valid = (instr\_opcode == 7'b1010011); // Detect FPU instructions
15. **Ensure PCPI Signals Work Correctly**
    * The pcpi\_fpu module must:
      + Accept operands (pcpi\_rs1, pcpi\_rs2).
      + Process the floating-point operation.
      + Return the result (pcpi\_rd).
      + Indicate completion (pcpi\_ready = 1).

**4️⃣ Modify the FPU to Work with PCPI**

* The fpu.v module should accept **PCPI signals** and return results accordingly.
* Ensure **opcode handling** matches the instruction format.
* Example: Modify the FPU to handle pcpi\_valid and pcpi\_ready properly:

always @(posedge clk or posedge reset) begin

if (reset) begin

pcpi\_ready <= 0;

pcpi\_wr <= 0;

end else if (pcpi\_valid) begin

pcpi\_rd <= fpu\_result;

pcpi\_ready <= 1;

pcpi\_wr <= 1;

end else begin

pcpi\_ready <= 0;

pcpi\_wr <= 0;

end

end

**5️⃣ Verify FPU Integration**

To ensure successful integration:  
✅ **Synthesize and simulate the design** in **ModelSim/QuestaSim**.  
✅ **Test with floating-point instructions** (fadd.s, fmul.s, fdiv.s, etc.).  
✅ **Check that the PCPI interface correctly routes instructions** to the FPU.  
✅ **Ensure correctness using a RISC-V software simulator (e.g., Spike, QEMU)**.

**Single Precision Floating Point Unit (FPU)**  
  
The given code is a hardware description for a **Single Precision Floating Point Unit (FPU)** based on the IEEE 754 standard, written in Verilog. The FPU supports addition, subtraction, multiplication, and division for single-precision floating-point numbers (32-bit format). Below is an explanation of the module and functionality.

Level Module: fpu

Purpose:

This is the main module that integrates arithmetic operations like addition, subtraction, multiplication, and division. It uses submodules for specific tasks and manages control logic to determine which operation to perform.

Inputs and Outputs

**input clk:** Clock signal for synchronization.

**input [31:0] A, B:** Operands in IEEE 754 single-precision format.

**input [1:0] opcode:** Operation code defining the arithmetic operation (ADD, SUB, MUL, DIV).

**output [31:0] O:** Result of the operation in IEEE 754 format.

**Intermediate Wires and Registers:**

a\_exponent, b\_exponent (8-bit): Extracted exponent fields of A and B.

a\_mantissa, b\_mantissa (24-bit): Extracted mantissa fields of A and B. Implicit leading 1 is added.

o\_sign, o\_exponent, o\_mantissa: Output components (sign, exponent, and mantissa).

Registers like adder\_a\_in, adder\_b\_in, etc., store operands for the respective operations.

Opcode Decoding

The opcode is decoded to determine the operation type:

**ADD:** Addition operation (opcode = 00).

**SUB:** Subtraction operation (opcode = 01).

**MUL:** Multiplication operation (opcode = 11).

**DIV:** Division operation (opcode = 10).

Submodule Instances:

**Adder (adder A1):** Performs floating-point addition.

**Multiplier (multiplier M1):** Performs floating-point multiplication.

**Divider (divider D1):** Uses a reciprocal approach for division.

Behavioral Logic (ALWAYS Block)

Based on opcode, the corresponding arithmetic operation is performed.

Special cases (like NaN, infinity, or zero) are handled explicitly.

The result is computed by invoking the respective arithmetic module, and the final IEEE 754 output is assembled using the o\_sign, o\_exponent, and o\_mantissa.

Arithmetic Modules

Adder Module (adder)

Handles addition and subtraction of IEEE 754 floating-point numbers.

**Step 1:** Extract Components

Sign: Extracted from the most significant bit.

Exponent: Retrieved from bits [30:23] with special handling for denormalized numbers.

Mantissa: Appends an implicit leading 1 if the number is normalized.

**Step 2:** Align Exponents

The mantissa of the smaller number is right-shifted to match the larger exponent.

**Step 3:** Add/Subtract Mantissas

Depending on the signs, mantissas are added (same signs) or subtracted (opposite signs).

**Step 4:** Normalize Result

Leading zeros are removed from the mantissa, and the exponent is adjusted.

Multiplier Module (multiplier)

Implements floating-point multiplication.

**Step 1:** Extract Components

Similar to the adder module.

**Step 2:** Compute Product

Mantissas are multiplied directly (48-bit product), and exponents are added with bias adjustment (a\_exponent + b\_exponent - 127).

**Step 3:** Normalize Product

If the leading bit of the product is 1, the product is shifted, and the exponent is incremented.

Divider Module (divider)

Performs division using the formula

Result=A×(1/B).

Reciprocal Calculation: An auxiliary module computes the reciprocal of B.

**Multiplication:** The reciprocal is multiplied by A using the multiplier module.

Normalization Modules

Addition Normalizer (addition\_normaliser)

Adjusts the mantissa and exponent after an addition or subtraction.

If leading bits of the mantissa are zero, the mantissa is shifted left, and the exponent is decreased.

Multiplication Normalizer (multiplication\_normaliser)

Adjusts the mantissa and exponent after multiplication.

Similar to the addition normalizer but operates on a 48-bit mantissa.

Reciprocal Module (reciprocal)

Calculates the approximate reciprocal of a floating-point number.

This is used in the division module for faster computation.

IEEE 754 Format Details

The 32-bit single-precision IEEE 754 format:

**1-bit Sign (S):** Determines the sign of the number.

**8-bit Exponent (E):** Encodes the exponent with a bias of 127.

**23-bit Mantissa (M):** Encodes the fractional part with an implicit leading 1 for normalized numbers.

Special Cases

Handled in all operations:

NaN (Not a Number):

Occurs when the exponent is 255 and the mantissa is non-zero.

Infinity:

Represented by an exponent of 255 and a zero mantissa.

Zero:

Special representation with zero exponent and mantissa.

# Installation of RISC-V GCC toolchain to run RISC-V PicoRV32 core

**1. Update and Install Prerequisites**

sudo apt update

sudo apt upgrade -y

sudo apt install build-essential gcc make git libgmp-dev libmpfr-dev libmpc-dev gawk bison flex texinfo gperf python3 python3-pip -y

**2. Clone and Build the RISC-V GCC Toolchain**

The RISC-V GNU Compiler Toolchain repository contains the necessary GCC for RISC-V:

# Clone the RISC-V GCC repository

git clone --recursive https://github.com/riscv-collab/riscv-gnu-toolchain

cd riscv-gnu-toolchain

# Build the GCC toolchain for RV32I (32-bit RISC-V ISA)

./configure --prefix=/opt/riscv --with-arch=rv32im --with-abi=ilp32

make -j$(nproc)

# Add the RISC-V GCC toolchain to your PATH

echo 'export PATH=/opt/riscv/bin:$PATH' >> ~/.bashrc

source ~/.bashrc

**3. Set Up PicoRV32**

Clone the PicoRV32 repository and test the setup:

git clone https://github.com/cliffordwolf/picorv32.git

cd picorv32

# Run the testbench to verify the build

make test

**4. Compile Your Code for PicoRV32**

Use the RISC-V GCC toolchain to compile code for the PicoRV32 core:

riscv32-unknown-elf-gcc -o your\_program.elf -march=rv32im -mabi=ilp32 your\_program.c

riscv32-unknown-elf-objcopy -O binary your\_program.elf your\_program.bin

**5. Run the Code in PicoRV32 Testbench**

Simulate the compiled program using the testbench provided in the PicoRV32 repository:

make firmware

make sim

This process should set up the GCC toolchain, build the PicoRV32 testbench, and allow you to simulate your programs. Let me know if you need additional steps!

**Step 1: Set Up the Environment**

Make sure you have:

* **Xilinx ISE Design Suite** (e.g., ISE 14.7) – the last official tool for Spartan-6
* Your **Spartan-6 development board** (e.g., Nexys3, Papilio Pro)
* USB JTAG cable & drivers
* Optional: A UART interface or 7-seg/LEDs for output testing

**🧾 Step 2: Create a New ISE Project**

1. **Open ISE Design Suite**
2. **Create New Project:**
   * File > New Project
   * Name: fpu\_test
   * Set the correct **Spartan-6 device** (e.g., xc6slx16-3-csg324)
3. Add your fpu.v and all submodules (fp\_add\_sub.v, fp\_multiply.v, etc.)
4. **Create a Top Module:** This is your test wrapper that connects the FPU to FPGA pins (you'll write this in step 5).

**🧪 Step 3: Simulate the FPU (Optional but Recommended)**

1. Create a testbench file (fpu\_tb.v)
2. Instantiate the FPU and drive it with test values
3. Use the ISE simulator (ISim) to check output correctness
4. Verify float operations behave as expected before moving to hardware

**🏗️ Step 4: Write the Top-Level Wrapper Module**

This will:

* Instantiate the FPU
* Provide input signals (e.g., via switches, buttons, or UART)
* Drive outputs to LEDs, 7-segment, or UART

Basic skeleton:

verilog

CopyEdit

module top (

input clk,

input rst,

input [3:0] opcode,

input [31:0] a,

input [31:0] b,

input load,

input store,

output [31:0] result

);

wire write\_enable = 1'b1; // Always write in this test

wire [31:0] mem\_data\_in = 32'b0; // Placeholder

wire [4:0] rd = 5'b00000; // Placeholder

wire [31:0] address = 32'b0;

fpu my\_fpu (

.a(a),

.b(b),

.opcode(opcode),

.load(load),

.store(store),

.mem\_data\_in(mem\_data\_in),

.rd(rd),

.write\_enable(write\_enable),

.address(address),

.result(result),

.mem\_data\_out()

);

endmodule

You may need to modify it to match your I/O device layout (switches, LEDs, etc.)

**🧱 Step 5: Assign Pins (UCF File)**

1. Create a .ucf file to map ports to physical FPGA pins.

Example for Nexys3:

ucf

CopyEdit

NET "clk" LOC = C9; # 50MHz clock

NET "rst" LOC = C10;

NET "a[0]" LOC = ...;

...

NET "result[0]" LOC = ...; # Connect to LEDs or serial output

Use your board’s reference manual to get correct pin locations.

**🛠️ Step 6: Implement Design**

1. Run:
   * Synthesize
   * Implement Design
   * Generate Programming File
2. Fix any synthesis or implementation errors

**🚀 Step 7: Program the FPGA**

1. Open **iMPACT**
2. Connect to your FPGA board via JTAG
3. Load the .bit file
4. Program the device

**🧪 Step 8: Verify on Hardware**

* Use UART, LEDs, or a 7-segment to verify results.
* Input test values via switches or a UART script.
* Confirm expected output from the FPU matches theoretical calculations.

**💡 Optional Enhancements:**

* Add UART input/output interface for test flexibility
* Add a simple FSM to step through multiple test cases
* Log output to serial console for debugging

If you'd like, I can help you:

* Write a testbench
* Write the UART module or FSM for control
* Generate a sample UCF for your board

Let me know which board you're using!

4o

**You said:**

testbench should test all included instructions

2/2

**ChatGPT said:**

Perfect! Here's a **Verilog testbench** that covers **all the included FPU instructions** in your fpu.v file:

* fadd.s, fsub.s
* fmul.s, fdiv.s, fsqrt.s
* fmin.s, fmax.s
* fcvt.w.s, fcvt.wu.s, fcvt.s.w, fcvt.s.wu
* feq.s, flt.s, fle.s

We'll use #10 delays between tests and hardcode a few test vectors using IEEE-754 encoding for single-precision floats.

**✅ Testbench: fpu\_tb.v**

verilog

CopyEdit

`timescale 1ns / 1ps

module fpu\_tb;

reg [31:0] a, b;

reg [3:0] opcode;

reg load, store, write\_enable;

reg [4:0] rd;

reg [31:0] address, mem\_data\_in;

wire [31:0] result;

wire [31:0] mem\_data\_out;

// Instantiate the FPU

fpu uut (

.a(a),

.b(b),

.opcode(opcode),

.load(load),

.store(store),

.mem\_data\_in(mem\_data\_in),

.rd(rd),

.write\_enable(write\_enable),

.address(address),

.result(result),

.mem\_data\_out(mem\_data\_out)

);

// Helper: real-to-hex conversion (for visual reference)

function [31:0] float\_to\_ieee;

input real val;

begin

float\_to\_ieee = $realtobits(val);

end

endfunction

initial begin

// Init

a = 0; b = 0; opcode = 0;

load = 0; store = 0; write\_enable = 1;

rd = 0; address = 0; mem\_data\_in = 0;

// Test fadd.s: 1.5 + 2.5 = 4.0

a = float\_to\_ieee(1.5);

b = float\_to\_ieee(2.5);

opcode = 4'b0000;

#10;

// Test fsub.s: 5.5 - 2.0 = 3.5

a = float\_to\_ieee(5.5);

b = float\_to\_ieee(2.0);

opcode = 4'b0001;

#10;

// Test fmul.s: 3.0 \* 2.0 = 6.0

a = float\_to\_ieee(3.0);

b = float\_to\_ieee(2.0);

opcode = 4'b0010;

#10;

// Test fdiv.s: 8.0 / 2.0 = 4.0

a = float\_to\_ieee(8.0);

b = float\_to\_ieee(2.0);

opcode = 4'b0011;

#10;

// Test fmin.s: min(1.2, 2.4) = 1.2

a = float\_to\_ieee(1.2);

b = float\_to\_ieee(2.4);

opcode = 4'b0100;

#10;

// Test fmax.s: max(1.2, 2.4) = 2.4

opcode = 4'b0101;

#10;

// Test fsqrt.s: sqrt(16.0) = 4.0

a = float\_to\_ieee(16.0);

opcode = 4'b0110;

#10;

// Test feq.s: 2.5 == 2.5 -> 1

a = float\_to\_ieee(2.5);

b = float\_to\_ieee(2.5);

opcode = 4'b1000;

#10;

// Test flt.s: 1.5 < 2.5 -> 1

a = float\_to\_ieee(1.5);

b = float\_to\_ieee(2.5);

opcode = 4'b1001;

#10;

// Test fle.s: 1.5 <= 1.5 -> 1

a = float\_to\_ieee(1.5);

b = float\_to\_ieee(1.5);

opcode = 4'b1010;

#10;

// Test fcvt.w.s: float 4.9 -> int 4

a = float\_to\_ieee(4.9);

opcode = 4'b1100;

#10;

// Test fcvt.wu.s: float 4.9 -> uint 4

opcode = 4'b1101;

#10;

// Test fcvt.s.w: int 5 -> float 5.0

a = 32'd5;

opcode = 4'b1110;

#10;

// Test fcvt.s.wu: uint 5 -> float 5.0

opcode = 4'b1111;

#10;

$finish;

end

endmodule

**🔍 Next Steps**

* Add fp\_add\_sub.v, fp\_multiply.v, etc., to your simulation project.
* Run this testbench using **ISim** or another Verilog simulator.
* Observe result output in the waveform or terminal.

Let me know if you want:

* Real-to-float printing helpers ($bitstoreal)
* Console display of output in human-readable format
* Version for test automation using Python + UART

Want me to bundle this with a full simulation setup?

4o

**You said:**

datasheet\_nexys3.pdf

PDF

This is my board datasheet read this and write UCF file so that i can test my designed fpu i will provide input in test bench and just output to be shown on the 7-segment display and leds

2/2

**ChatGPT said:**

Thanks for sharing the datasheet for the Nexys 3 board. Based on your request, I'll generate a UCF (User Constraints File) so that:

* You can input data via your testbench.
* The output of your FPU (Floating Point Unit) will be visible on the 7-segment displays and LEDs.

Assuming the following:

* You're driving the output of your FPU to 8 LEDs (LD0–LD7) and one or more 7-segment displays (CA–CG, DP, and AN0–AN3).
* You will manage segment multiplexing and decoding in your design.
* The clock and reset pins are optional or managed internally via simulation/testbench.

Here's a basic UCF example for Nexys 3:

ucf

CopyEdit

# Clock (if needed)

NET "clk" LOC = "E3" | IOSTANDARD = LVCMOS33; # 100 MHz clock

# LEDs (LD0–LD7)

NET "led[0]" LOC = "T1" | IOSTANDARD = LVCMOS33;

NET "led[1]" LOC = "R1" | IOSTANDARD = LVCMOS33;

NET "led[2]" LOC = "P1" | IOSTANDARD = LVCMOS33;

NET "led[3]" LOC = "N1" | IOSTANDARD = LVCMOS33;

NET "led[4]" LOC = "T3" | IOSTANDARD = LVCMOS33;

NET "led[5]" LOC = "T2" | IOSTANDARD = LVCMOS33;

NET "led[6]" LOC = "R3" | IOSTANDARD = LVCMOS33;

NET "led[7]" LOC = "R2" | IOSTANDARD = LVCMOS33;

# 7-Segment Display segments (CA–CG, DP)

NET "seg[0]" LOC = "L14" | IOSTANDARD = LVCMOS33; # CA

NET "seg[1]" LOC = "H12" | IOSTANDARD = LVCMOS33; # CB

NET "seg[2]" LOC = "N14" | IOSTANDARD = LVCMOS33; # CC

NET "seg[3]" LOC = "N11" | IOSTANDARD = LVCMOS33; # CD

NET "seg[4]" LOC = "P12" | IOSTANDARD = LVCMOS33; # CE

NET "seg[5]" LOC = "L13" | IOSTANDARD = LVCMOS33; # CF

NET "seg[6]" LOC = "M12" | IOSTANDARD = LVCMOS33; # CG

NET "seg[7]" LOC = "N13" | IOSTANDARD = LVCMOS33; # DP

# 7-Segment Display anodes (AN0–AN3)

NET "an[0]" LOC = "M13" | IOSTANDARD = LVCMOS33;

NET "an[1]" LOC = "J12" | IOSTANDARD = LVCMOS33;

NET "an[2]" LOC = "F12" | IOSTANDARD = LVCMOS33;

NET "an[3]" LOC = "D14" | IOSTANDARD = LVCMOS33;  
  
  
  
  
  
module fpu (

    input [31:0] a, b,        // Floating-point inputs

    input [31:0] mem\_data\_in, // Data from memory (for load instruction)

    input [3:0] opcode,       // Operation selector (expanded to 4 bits to accommodate all operations)

    input load,               // Load control signal

    input store,              // Store control signal

    input [31:0] address,     // Address for load/store operations

    input [4:0] rd,           // Destination register index (0-31)

    input write\_enable,       // Write enable signal for registers

    output reg [31:0] result, // Computed result

    output reg [31:0] mem\_data\_out // Data to store in memory

);

    // Floating-Point Register File

    reg [31:0] f[31:0];  // 32 Floating-point registers f0-f31

    reg [31:0] fcsr;     // Floating-Point Control and Status Register (fcsr)

    wire [31:0] add\_sub\_result, mul\_result, div\_result, fmin\_result, fmax\_result, sqrt\_result;

    wire [31:0] cvt\_w\_result, cvt\_s\_result;

    wire feq\_result, flt\_result, fle\_result;

    // Floating Point Operations

    fp\_add\_sub adder\_subtractor (.a(a), .b(b), .add\_sub(opcode[0]), .result(add\_sub\_result));

    fp\_multiply multiplier (.a(a), .b(b), .result(mul\_result));

    fp\_divide divider (.a(a), .b(b), .result(div\_result));

    fp\_minmax min\_max\_unit (.a(a), .b(b), .op(opcode[0]), .result(fmin\_result));  // MIN/MAX operation

    fp\_sqrt sqrt\_unit (.a(a), .result(sqrt\_result));  // SQRT operation

    fp\_compare compare\_unit (.a(a), .b(b), .op(opcode[1:0]), .result\_bit(feq\_result), .result\_lt(flt\_result), .result\_le(fle\_result));

    // New Conversion Operations

    fp\_to\_int fp\_to\_int\_unit (.a(a), .unsigned\_flag(opcode[0]), .result(cvt\_w\_result));  // FCVT.W.S/FCVT.WU.S

    int\_to\_fp int\_to\_fp\_unit (.a(a), .unsigned\_flag(opcode[0]), .result(cvt\_s\_result));  // FCVT.S.W/FCVT.S.WU

    // Operation Selection

    always @(\*) begin

        case (opcode)

            4'b0000: result = add\_sub\_result;       // ADD

            4'b0001: result = add\_sub\_result;       // SUB

            4'b0010: result = mul\_result;           // MUL

            4'b0011: result = div\_result;           // DIV

            4'b0100: result = fmin\_result;          // FMIN.S

            4'b0101: result = fmax\_result;          // FMAX.S

            4'b0110: result = sqrt\_result;          // FSQRT.S

            4'b1000: result = {31'b0, feq\_result};  // FEQ.S

            4'b1001: result = {31'b0, flt\_result};  // FLT.S

            4'b1010: result = {31'b0, fle\_result};  // FLE.S

            4'b1100: result = cvt\_w\_result;         // FCVT.W.S (signed int conversion)

            4'b1101: result = cvt\_w\_result;         // FCVT.WU.S (unsigned int conversion)

            4'b1110: result = cvt\_s\_result;         // FCVT.S.W (signed int to float)

            4'b1111: result = cvt\_s\_result;         // FCVT.S.WU (unsigned int to float)

            default: result = 32'h00000000;

        endcase

    end

    // Load/Store Operations

    always @(\*) begin

        if (load) begin

            if (address[1:0] != 2'b00) fcsr[3] = 1; // Misalignment exception

            else result = mem\_data\_in;

        end

        if (store) begin

            if (address[1:0] != 2'b00) fcsr[3] = 1; // Misalignment exception

            else mem\_data\_out = a;

        end

    end

    // Register Write Mechanism

    always @(posedge write\_enable) begin

        f[rd] <= result;  // Write result to selected floating-point register

    end

endmodule

module fpu (

    input [31:0] a, b,        // Floating-point inputs

    input [31:0] mem\_data\_in, // Data from memory (for load instruction)

    input [3:0] opcode,       // Operation selector (expanded to 4 bits to accommodate new operations)

    input load,               // Load control signal

    input store,              // Store control signal

    input [31:0] address,     // Address for load/store operations

    input [4:0] rd,           // Destination register index (0-31)

    input write\_enable,       // Write enable signal for registers

    output reg [31:0] result, // Computed result

    output reg [31:0] mem\_data\_out // Data to store in memory

);

    // Floating-Point Register File

    reg [31:0] f[31:0];  // 32 Floating-point registers f0-f31

    reg [31:0] fcsr;     // Floating-Point Control and Status Register (fcsr)

    wire [31:0] add\_sub\_result, mul\_result, div\_result, fmin\_result, fmax\_result, sqrt\_result;

    wire feq\_result, flt\_result, fle\_result;

    // Floating Point Operations

    fp\_add\_sub adder\_subtractor (.a(a), .b(b), .add\_sub(opcode[0]), .result(add\_sub\_result));

    fp\_multiply multiplier (.a(a), .b(b), .result(mul\_result));

    fp\_divide divider (.a(a), .b(b), .result(div\_result));

    fp\_minmax min\_max\_unit (.a(a), .b(b), .op(opcode[0]), .result(fmin\_result));  // MIN/MAX operation

    fp\_sqrt sqrt\_unit (.a(a), .result(sqrt\_result));  // SQRT operation

    fp\_compare compare\_unit (.a(a), .b(b), .op(opcode[1:0]), .result\_bit(feq\_result), .result\_lt(flt\_result), .result\_le(fle\_result));

    // Operation Selection

    always @(\*) begin

        case (opcode)

            4'b0000: result = add\_sub\_result;       // ADD

            4'b0001: result = add\_sub\_result;       // SUB

            4'b0010: result = mul\_result;           // MUL

            4'b0011: result = div\_result;           // DIV

            4'b0100: result = fmin\_result;          // FMIN.S

            4'b0101: result = fmax\_result;          // FMAX.S

            4'b0110: result = sqrt\_result;          // FSQRT.S

            4'b1000: result = {31'b0, feq\_result};  // FEQ.S

            4'b1001: result = {31'b0, flt\_result};  // FLT.S

            4'b1010: result = {31'b0, fle\_result};  // FLE.S

            default: result = 32'h00000000;

        endcase

    end

    // Load/Store Operations

    always @(\*) begin

        if (load) begin

            if (address[1:0] != 2'b00) fcsr[3] = 1; // Misalignment exception

            else result = mem\_data\_in;

        end

        if (store) begin

            if (address[1:0] != 2'b00) fcsr[3] = 1; // Misalignment exception

            else mem\_data\_out = a;

        end

    end

    // Register Write Mechanism

    always @(posedge write\_enable) begin

        f[rd] <= result;  // Write result to selected floating-point register

    end

endmodule

// Floating-Point Adder/Subtractor (Unchanged from original)

module fp\_add\_sub (

    input [31:0] a, b,

    input add\_sub,  // 0 for addition, 1 for subtraction

    output reg [31:0] result

);

    wire sign\_a = a[31];

    wire sign\_b = b[31] ^ add\_sub;  // Flip sign for subtraction

    wire [7:0] exp\_a = a[30:23], exp\_b = b[30:23];

    wire [23:0] mant\_a = {1'b1, a[22:0]};  // Implicit leading 1

    wire [23:0] mant\_b = {1'b1, b[22:0]};

    wire exp\_greater = (exp\_a > exp\_b);

    wire [7:0] exp\_diff = exp\_greater ? (exp\_a - exp\_b) : (exp\_b - exp\_a);

    wire [23:0] mant\_a\_shifted = exp\_greater ? mant\_a : (mant\_a >> exp\_diff);

    wire [23:0] mant\_b\_shifted = exp\_greater ? (mant\_b >> exp\_diff) : mant\_b;

    wire [23:0] mant\_large = exp\_greater ? mant\_a : mant\_b;

    wire [23:0] mant\_small = exp\_greater ? mant\_b\_shifted : mant\_a\_shifted;

    wire [7:0] exp\_large = exp\_greater ? exp\_a : exp\_b;

    wire sign\_large = exp\_greater ? sign\_a : sign\_b;

    wire sign\_small = exp\_greater ? sign\_b : sign\_a;

    wire [24:0] mant\_sum = (sign\_large == sign\_small) ?

                           (mant\_large + mant\_small) :

                           (mant\_large - mant\_small);

    reg [7:0] exp\_norm;

    reg [23:0] mant\_norm;

    reg sign\_result;

    integer shift\_count;

    always @(\*) begin

        if (mant\_sum[24]) begin

            // Overflow occurred, shift right

            exp\_norm = exp\_large + 1;

            mant\_norm = mant\_sum[24:1];

        end else begin

            // Normalize by shifting left if necessary

            mant\_norm = mant\_sum[23:0];

            exp\_norm = exp\_large;

            shift\_count = 0;

            while (mant\_norm[23] == 0 && exp\_norm > 0) begin

                mant\_norm = mant\_norm << 1;

                exp\_norm = exp\_norm - 1;

                shift\_count = shift\_count + 1;

            end

        end

        // Determine final sign

        if ((sign\_large != sign\_small) && (mant\_large < mant\_small)) begin

            sign\_result = sign\_small;  // Result takes the sign of the larger absolute value

            mant\_norm = (~mant\_norm + 1);  // Two's complement for negative results

        end else begin

            sign\_result = sign\_large;

        end

        result = {sign\_result, exp\_norm, mant\_norm[22:0]};

    end

endmodule

// Floating-Point Multiplier (Unchanged from original)

module fp\_multiply (

    input [31:0] a, b,

    output reg [31:0] result

);

    wire sign\_a = a[31], sign\_b = b[31];

    wire [7:0] exp\_a = a[30:23], exp\_b = b[30:23];

    wire [23:0] mant\_a = {1'b1, a[22:0]}, mant\_b = {1'b1, b[22:0]};

    wire sign\_result = sign\_a ^ sign\_b;

    wire [8:0] exp\_result = exp\_a + exp\_b - 127;

    wire [47:0] mant\_result = mant\_a \* mant\_b;

    reg [22:0] mant\_norm;

    reg [7:0] exp\_norm;

    always @(\*) begin

        if (mant\_result[47]) begin

            mant\_norm = mant\_result[46:24];

            exp\_norm = exp\_result + 1;

        end else begin

            mant\_norm = mant\_result[45:23];

            exp\_norm = exp\_result;

        end

        result = {sign\_result, exp\_norm, mant\_norm};

    end

endmodule

// Floating-Point Divider (Unchanged from original)

module fp\_divide (

    input [31:0] a, b,

    output reg [31:0] result

);

    wire sign\_a = a[31], sign\_b = b[31];

    wire [7:0] exp\_a = a[30:23], exp\_b = b[30:23];

    wire [23:0] mant\_a = {1'b1, a[22:0]}, mant\_b = {1'b1, b[22:0]};

    wire sign\_result = sign\_a ^ sign\_b;

    wire [8:0] exp\_result = exp\_a - exp\_b + 127;

    wire [47:0] mant\_result = (mant\_a << 23) / mant\_b;

    reg [22:0] mant\_norm;

    reg [7:0] exp\_norm;

    always @(\*) begin

        mant\_norm = mant\_result[22:0];

        exp\_norm = exp\_result;

        result = (b == 32'h00000000) ? 32'h7fc00000 : {sign\_result, exp\_norm, mant\_norm};

    end

endmodule

// Floating-Point Minimum and Maximum (FMIN.S, FMAX.S) - Unchanged from original

module fp\_minmax (

    input [31:0] a, b,

    input op,  // 0 for FMIN, 1 for FMAX

    output reg [31:0] result

);

    function is\_nan;

        input [31:0] x;

        is\_nan = (x[30:23] == 8'hFF) && (x[22:0] != 0);

    endfunction

    always @(\*) begin

        if (is\_nan(a) && is\_nan(b)) begin

            result = 32'h7FC00000; // Canonical NaN

        end else if (is\_nan(a)) begin

            result = b;  // If only `a` is NaN, return `b`

        end else if (is\_nan(b)) begin

            result = a;  // If only `b` is NaN, return `a`

        end else if (op == 1) begin

            if ((a[31] == 0) && (b[31] == 0)) begin

              if(a[30:23] == b[30:23])begin

                if(a[22:0] == b[22:0])begin

                  result = a;

                end else if (a[22:0] > b[22:0]) begin

                     result = a;

                end else if (a[22:0] < b[22:0]) begin

                     result = b;

                 end

              end else if(a[30:23] > b[30:23])begin

                result=a;

                 end else if(a[30:23] < b[30:23])begin

                   result=b;

              end

            end else if ((a[31] == 1) && (b[31] == 1))begin

              if(a[30:23] == b[30:23])begin

                if(a[22:0] == b[22:0])begin

                  result = a;

                end else if (a[22:0] > b[22:0]) begin

                     result = b;

                end else if (a[22:0] < b[22:0]) begin

                     result = a;

                 end

              end else if(a[30:23] > b[30:23])begin

                result=b;

                 end else if(a[30:23] < b[30:23])begin

                   result=a;

              end

            end else if ((a[31] == 1) && (b[31] == 0) ) begin

              result=b;

            end else if ((a[31] == 0) && (b[31] == 1) ) begin

              result=a;

            end else begin

              result = 32'h7FC00000; // Canonical NaN

            end

        end else if (op == 0) begin

          if ((a[31] == 0) && (b[31] == 0)) begin

              if(a[30:23] == b[30:23])begin

                if(a[22:0] == b[22:0])begin

                  result = a;

                end else if (a[22:0] > b[22:0]) begin

                     result = b;

                end else if (a[22:0] < b[22:0]) begin

                     result = a;

                 end

              end else if(a[30:23] > b[30:23])begin

                result=b;

                 end else if(a[30:23] < b[30:23])begin

                   result=a;

              end

          end else if ((a[31] == 1) && (b[31] == 1) ) begin

              if(a[30:23] == b[30:23])begin

                if(a[22:0] == b[22:0])begin

                  result = a;

                end else if (a[22:0] > b[22:0]) begin

                     result = a;

                end else if (a[22:0] < b[22:0]) begin

                     result = b;

                 end

              end else if(a[30:23] > b[30:23])begin

                result=a;

                 end else if(a[30:23] < b[30:23])begin

                   result=b;

              end

          end else if ((a[31] == 1) && (b[31] == 0) ) begin

              result=a;

          end else if ((a[31] == 0) && (b[31] == 1) ) begin

              result=b;

          end else begin

              result = 32'h7FC00000; // Canonical NaN

          end

        end else begin

          result = 32'h7FC00000;

        end

    end

endmodule

// NEW MODULE: Floating-Point Square Root (FSQRT.S)

module fp\_sqrt (

    input [31:0] a,

    output reg [31:0] result

);

    // Extract components

    wire sign\_a = a[31];

    wire [7:0] exp\_a = a[30:23];

    wire [22:0] frac\_a = a[22:0];

    // Square root specific variables

    reg [7:0] exp\_result;

    reg [22:0] frac\_result;

    reg [24:0] operand;  // For normalized mantissa with guard bits

    reg [24:0] root;     // The computed square root value

    reg [24:0] remainder;

    reg [24:0] temp;

    integer i;

    // Handle special cases

    wire is\_zero = (exp\_a == 0) && (frac\_a == 0);

    wire is\_inf = (exp\_a == 8'hFF) && (frac\_a == 0);

    wire is\_nan = (exp\_a == 8'hFF) && (frac\_a != 0);

    wire is\_neg = sign\_a && !is\_zero;  // Negative and not zero

    always @(\*) begin

        // Special cases handling

        if (is\_zero) begin

            // Square root of zero is zero

            result = a;  // Preserve sign of zero

        end else if (is\_inf && !sign\_a) begin

            // Square root of +infinity is +infinity

            result = a;

        end else if (is\_nan || is\_neg || (is\_inf && sign\_a)) begin

            // Square root of NaN, negative number, or -infinity is NaN

            result = 32'h7FC00000;  // canonical NaN

        end else begin

            // Normal computation for positive finite numbers

            // Prepare the operand - normalize the mantissa

            operand = {1'b1, frac\_a, 1'b0};  // Include implicit 1 and guard bit

            // Adjust exponent

            if (exp\_a[0]) begin  // Odd exponent

                operand = {operand[23:0], 1'b0};  // Left shift by 1 for normalization

            end

            // Final exponent is half of the original (minus bias adjustment)

            exp\_result = ((exp\_a - 127) >> 1) + 127;

            // Non-restoring square root algorithm

            root = 0;

            remainder = 0;

            for (i = 0; i < 24; i = i + 1) begin

                remainder = {remainder[22:0], operand[24-i], operand[23-i]};

                temp = {root, 1'b1};

                if (remainder >= temp) begin

                    remainder = remainder - temp;

                    root = {root[22:0], 1'b1};

                end else begin

                    root = {root[22:0], 1'b0};

                end

            end

            // Round the result (simplified round-to-nearest)

            if (remainder > 0) begin

                root = root + 1;  // Round up if there's a remainder

            end

            // Normalize the result if needed

            if (root[23]) begin

                frac\_result = root[22:0];

            end else begin

                frac\_result = root[21:0];  // Shift left if leading bit is zero

                exp\_result = exp\_result - 1;

            end

            // Assemble the final result

            result = {1'b0, exp\_result, frac\_result};

        end

    end

endmodule

// NEW MODULE: Floating-Point Compare (FEQ.S, FLT.S, FLE.S)

module fp\_compare (

    input [31:0] a, b,

    input [1:0] op,  // 00: FEQ, 01: FLT, 10: FLE

    output reg result\_bit, // General result bit

    output reg result\_lt,  // Less than result

    output reg result\_le   // Less than or equal result

);

    // Check for NaN values

    function is\_nan;

        input [31:0] x;

        is\_nan = (x[30:23] == 8'hFF) && (x[22:0] != 0);

    endfunction

    // Check for equality

    wire equal = (a == b) || (is\_zero(a) && is\_zero(b));  // Consider both +0 and -0 as equal

    // Check for zero value

    function is\_zero;

        input [31:0] x;

        is\_zero = (x[30:23] == 0) && (x[22:0] == 0);  // Exponent and fraction are zero

    endfunction

    // Less than comparison logic

    wire a\_negative = a[31] && !(is\_zero(a));  // Negative and not zero

    wire b\_negative = b[31] && !(is\_zero(b));  // Negative and not zero

    wire less\_than = (a\_negative && !b\_negative) ||                                   // a negative, b positive

                    (a\_negative && b\_negative && (a[30:0] > b[30:0])) ||              // both negative, |a| > |b|

                    (!a\_negative && !b\_negative && (a[30:0] < b[30:0]));              // both positive, a < b

    always @(\*) begin

        if (is\_nan(a) || is\_nan(b)) begin

            // NaN values make all comparisons return false

            result\_bit = 0;

            result\_lt = 0;

            result\_le = 0;

        end else begin

            // Equal comparison

            result\_bit = equal;

            // Less than comparison

            result\_lt = less\_than;

            // Less than or equal comparison

            result\_le = equal || less\_than;

        end

    end

endmodule

// Floating-Point to Integer Conversion Module (FCVT.W.S, FCVT.WU.S)

module fp\_to\_int (

    input [31:0] a,           // Floating-point input

    input unsigned\_flag,      // 0 for signed (FCVT.W.S), 1 for unsigned (FCVT.WU.S)

    output reg [31:0] result  // Integer result

);

    // Extract components

    wire sign\_a = a[31];

    wire [7:0] exp\_a = a[30:23];

    wire [22:0] frac\_a = a[22:0];

    wire [23:0] mant\_a = {1'b1, frac\_a}; // Implicit leading 1

    // Special cases

    wire is\_zero = (exp\_a == 0) && (frac\_a == 0);

    wire is\_inf = (exp\_a == 8'hFF) && (frac\_a == 0);

    wire is\_nan = (exp\_a == 8'hFF) && (frac\_a != 0);

    // Variables for conversion

    reg [31:0] int\_value;

    reg [7:0] shift\_amount;

    always @(\*) begin

        // Handle special cases

        if (is\_zero) begin

            result = 32'h00000000;  // Zero maps to zero

        end

        else if (is\_nan || is\_inf) begin

            if (unsigned\_flag) begin

                result = (sign\_a && !is\_nan) ? 32'h00000000 : 32'hFFFFFFFF;  // -Inf -> 0, +Inf/NaN -> max unsigned

            else

                result = (sign\_a && !is\_nan) ? 32'h80000000 : 32'h7FFFFFFF;  // -Inf -> min signed, +Inf/NaN -> max signed

            end

        end

        else begin

            // Calculate unbiased exponent

            shift\_amount = exp\_a - 127;

            if (shift\_amount > 31) begin

                // Overflow cases

                if (unsigned\_flag) begin

                    result = (sign\_a) ? 32'h00000000 : 32'hFFFFFFFF;  // Negative -> 0, Positive -> max unsigned

                end else begin

                    result = (sign\_a) ? 32'h80000000 : 32'h7FFFFFFF;  // Negative -> min signed, Positive -> max signed

                end

            end

            else if (shift\_amount < 0) begin

                // Fractional numbers between -1 and 1

                result = 32'h00000000;  // Truncate to zero

            end

            else begin

                // Normal conversion

                if (shift\_amount <= 23) begin

                    // Shift mantissa according to exponent value

                    int\_value = mant\_a >> (23 - shift\_amount);

                end else begin

                    // Need to shift left for large exponents

                    int\_value = mant\_a << (shift\_amount - 23);

                end

                // Handle sign for signed integers

                if (sign\_a) begin

                    if (unsigned\_flag)

                        result = 32'h00000000;  // Negative float to unsigned int is 0

                    else

                        result = (~int\_value + 1);  // Two's complement for negative

                end else begin

                    result = int\_value;  // Positive value stays the same

                end

            end

        end

    end

endmodule

// Integer to Floating-Point Conversion Module (FCVT.S.W, FCVT.S.WU)

module int\_to\_fp (

    input [31:0] a,           // Integer input

    input unsigned\_flag,      // 0 for signed (FCVT.S.W), 1 for unsigned (FCVT.S.WU)

    output reg [31:0] result  // Floating-point result

);

    reg sign\_result;

    reg [7:0] exp\_result;

    reg [22:0] frac\_result;

    reg [31:0] abs\_value;

    integer leading\_zeros;

    always @(\*) begin

        // Handle sign

        if (unsigned\_flag) begin

            sign\_result = 1'b0;  // Always positive for unsigned

            abs\_value = a;

        end else begin

            sign\_result = a[31];  // Sign bit of signed integer

            abs\_value = sign\_result ? (~a + 1) : a;  // Two's complement if negative

        end

        // Handle zero special case

        if (abs\_value == 0) begin

            result = 32'h00000000;  // +0.0

        end else begin

            // Count leading zeros to normalize

            leading\_zeros = 0;

            if (abs\_value[31:16] == 0) begin leading\_zeros = leading\_zeros + 16; abs\_value = abs\_value << 16; end

            if (abs\_value[31:24] == 0) begin leading\_zeros = leading\_zeros + 8; abs\_value = abs\_value << 8; end

            if (abs\_value[31:28] == 0) begin leading\_zeros = leading\_zeros + 4; abs\_value = abs\_value << 4; end

            if (abs\_value[31:30] == 0) begin leading\_zeros = leading\_zeros + 2; abs\_value = abs\_value << 2; end

            if (abs\_value[31] == 0) begin leading\_zeros = leading\_zeros + 1; abs\_value = abs\_value << 1; end

            // Calculate exponent (biased)

            exp\_result = 8'd127 + 8'd31 - leading\_zeros;

            // Calculate fraction (mantissa without implied 1)

            frac\_result = abs\_value[30:8];

            // Handle rounding (simplified round-to-nearest)

            if (abs\_value[7]) begin

                // Check if we need to round up

                if (frac\_result == 23'h7FFFFF) begin

                    // If all fraction bits are 1, rounding up would cause overflow

                    frac\_result = 0;

                    exp\_result = exp\_result + 1;

                end else begin

                    frac\_result = frac\_result + 1;

                end

            end

            // Construct result

            result = {sign\_result, exp\_result, frac\_result};

        end

    end

endmodule

`timescale 1ns / 1ps

module fpu\_tb;

    // Inputs

    reg [31:0] a, b;

    reg [31:0] mem\_data\_in;

    reg [3:0] opcode;

    reg load, store;

    reg [31:0] address;

    reg [4:0] rd;

    reg write\_enable;

    // Outputs

    wire [31:0] result;

    wire [31:0] mem\_data\_out;

    // Instantiate the FPU

    fpu dut (

        .a(a),

        .b(b),

        .mem\_data\_in(mem\_data\_in),

        .opcode(opcode),

        .load(load),

        .store(store),

        .address(address),

        .rd(rd),

        .write\_enable(write\_enable),

        .result(result),

        .mem\_data\_out(mem\_data\_out)

    );

    // Define special IEEE-754 values

    localparam [31:0]

        ZERO\_POS       = 32'h00000000,    // +0.0

        ZERO\_NEG       = 32'h80000000,    // -0.0

        INF\_POS        = 32'h7F800000,    // +Infinity

        INF\_NEG        = 32'hFF800000,    // -Infinity

        NAN\_CANONICAL  = 32'h7FC00000,    // Canonical NaN

        ONE\_POS        = 32'h3F800000,    // +1.0

        ONE\_NEG        = 32'hBF800000,    // -1.0

        TWO\_POS        = 32'h40000000,    // +2.0

        HALF\_POS       = 32'h3F000000,    // +0.5

        QUARTER\_POS    = 32'h3E800000,    // +0.25

        PI             = 32'h40490FDB,    // +3.14159...

        E              = 32'h402DF854,    // +2.71828...

        SMALL\_POS      = 32'h00800000,    // Very small positive number

        LARGE\_POS      = 32'h7F7FFFFF;    // Very large positive number

    // Convert IEEE-754 to real for display purposes

    function real ieee\_to\_real;

        input [31:0] ieee;

        reg sign;

        reg [7:0] exp;

        reg [22:0] frac;

        real result;

        begin

            sign = ieee[31];

            exp = ieee[30:23];

            frac = ieee[22:0];

            if (exp == 0) begin

                // Denormalized numbers or zero

                if (frac == 0)

                    result = 0.0;

                else

                    result = $itor(frac) / $pow(2, 23) \* $pow(2, -126);

            end

            else if (exp == 255) begin

                // Infinity or NaN

                if (frac == 0)

                    result = sign ? -1.0/0.0 : 1.0/0.0;

                else

                    result = 0.0/0.0; // NaN

            end

            else begin

                // Normalized numbers

                result = (1.0 + $itor(frac) / $pow(2, 23)) \* $pow(2, $signed(exp - 127));

            end

            if (sign)

                result = -result;

            ieee\_to\_real = result;

        end

    endfunction

    // Helper function to display test results

    task display\_result;

        input [31:0] a\_val, b\_val, result\_val;

        input [3:0] op;

        input pass;

        reg [64\*8:0] op\_str;

        begin

            case(op)

                4'b0000: op\_str = "FADD.S";

                4'b0001: op\_str = "FSUB.S";

                4'b0010: op\_str = "FMUL.S";

                4'b0011: op\_str = "FDIV.S";

                4'b0100: op\_str = "FMIN.S";

                4'b0101: op\_str = "FMAX.S";

                4'b0110: op\_str = "FSQRT.S";

                4'b1000: op\_str = "FEQ.S";

                4'b1001: op\_str = "FLT.S";

                4'b1010: op\_str = "FLE.S";

                default: op\_str = "UNKNOWN";

            endcase

            $display("Test %s: a = %h (%f), b = %h (%f), result = %h (%f) - %s",

                     op\_str,

                     a\_val, ieee\_to\_real(a\_val),

                     b\_val, ieee\_to\_real(b\_val),

                     result\_val,

                     (op >= 4'b1000) ? $itor(result\_val[0]) : ieee\_to\_real(result\_val),

                     pass ? "PASS" : "FAIL");

        end

    endtask

    // Test vectors and expected results

    reg [31:0] expected\_result;

    reg test\_pass;

    // Main test procedure

    initial begin

        // Initialize

        a = 0;

        b = 0;

        mem\_data\_in = 0;

        opcode = 0;

        load = 0;

        store = 0;

        address = 0;

        rd = 0;

        write\_enable = 0;

        $display("\n--- FPU Testbench - Testing All Instructions ---\n");

        // Allow some time for initialization

        #10;

        // ----------------------------------------------------------------

        // Test 1: Addition (FADD.S)

        // ----------------------------------------------------------------

        $display("\nTesting FADD.S Instruction:");

        opcode = 4'b0000;

        // Test Case 1: 1.0 + 2.0 = 3.0

        a = ONE\_POS;

        b = TWO\_POS;

        expected\_result = 32'h40400000; // 3.0

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 2: 1.0 + (-1.0) = 0.0

        a = ONE\_POS;

        b = ONE\_NEG;

        expected\_result = ZERO\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 3: 0.0 + 0.0 = 0.0

        a = ZERO\_POS;

        b = ZERO\_POS;

        expected\_result = ZERO\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 4: Inf + 1.0 = Inf

        a = INF\_POS;

        b = ONE\_POS;

        expected\_result = INF\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 5: NaN + 1.0 = NaN

        a = NAN\_CANONICAL;

        b = ONE\_POS;

        #10;

        test\_pass = ((result[30:23] == 8'hFF) && (result[22] == 1)); // Check if result is NaN

        display\_result(a, b, result, opcode, test\_pass);

        // ----------------------------------------------------------------

        // Test 2: Subtraction (FSUB.S)

        // ----------------------------------------------------------------

        $display("\nTesting FSUB.S Instruction:");

        opcode = 4'b0001;

        // Test Case 1: 2.0 - 1.0 = 1.0

        a = TWO\_POS;

        b = ONE\_POS;

        expected\_result = ONE\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 2: 1.0 - 1.0 = 0.0

        a = ONE\_POS;

        b = ONE\_POS;

        expected\_result = ZERO\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 3: 0.0 - 1.0 = -1.0

        a = ZERO\_POS;

        b = ONE\_POS;

        expected\_result = ONE\_NEG;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // ----------------------------------------------------------------

        // Test 3: Multiplication (FMUL.S)

        // ----------------------------------------------------------------

        $display("\nTesting FMUL.S Instruction:");

        opcode = 4'b0010;

        // Test Case 1: 2.0 \* 1.0 = 2.0

        a = TWO\_POS;

        b = ONE\_POS;

        expected\_result = TWO\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 2: 0.5 \* 0.5 = 0.25

        a = HALF\_POS;

        b = HALF\_POS;

        expected\_result = QUARTER\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 3: 1.0 \* 0.0 = 0.0

        a = ONE\_POS;

        b = ZERO\_POS;

        expected\_result = ZERO\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 4: 1.0 \* Inf = Inf

        a = ONE\_POS;

        b = INF\_POS;

        expected\_result = INF\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // ----------------------------------------------------------------

        // Test 4: Division (FDIV.S)

        // ----------------------------------------------------------------

        $display("\nTesting FDIV.S Instruction:");

        opcode = 4'b0011;

        // Test Case 1: 2.0 / 1.0 = 2.0

        a = TWO\_POS;

        b = ONE\_POS;

        expected\_result = TWO\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 2: 1.0 / 2.0 = 0.5

        a = ONE\_POS;

        b = TWO\_POS;

        expected\_result = HALF\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 3: 0.0 / 1.0 = 0.0

        a = ZERO\_POS;

        b = ONE\_POS;

        expected\_result = ZERO\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 4: 1.0 / 0.0 = Inf

        a = ONE\_POS;

        b = ZERO\_POS;

        expected\_result = INF\_POS;

        #10;

        test\_pass = ((result[30:23] == 8'hFF) && (result[22:0] == 0)); // Check if result is Inf

        display\_result(a, b, result, opcode, test\_pass);

        // ----------------------------------------------------------------

        // Test 5: Minimum (FMIN.S)

        // ----------------------------------------------------------------

        $display("\nTesting FMIN.S Instruction:");

        opcode = 4'b0100;

        // Test Case 1: min(1.0, 2.0) = 1.0

        a = ONE\_POS;

        b = TWO\_POS;

        expected\_result = ONE\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 2: min(-1.0, 1.0) = -1.0

        a = ONE\_NEG;

        b = ONE\_POS;

        expected\_result = ONE\_NEG;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 3: min(NaN, 1.0) = 1.0

        a = NAN\_CANONICAL;

        b = ONE\_POS;

        expected\_result = ONE\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // ----------------------------------------------------------------

        // Test 6: Maximum (FMAX.S)

        // ----------------------------------------------------------------

        $display("\nTesting FMAX.S Instruction:");

        opcode = 4'b0101;

        // Test Case 1: max(1.0, 2.0) = 2.0

        a = ONE\_POS;

        b = TWO\_POS;

        expected\_result = TWO\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 2: max(-1.0, 1.0) = 1.0

        a = ONE\_NEG;

        b = ONE\_POS;

        expected\_result = ONE\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 3: max(NaN, 1.0) = 1.0

        a = NAN\_CANONICAL;

        b = ONE\_POS;

        expected\_result = ONE\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // ----------------------------------------------------------------

        // Test 7: Square Root (FSQRT.S)

        // ----------------------------------------------------------------

        $display("\nTesting FSQRT.S Instruction:");

        opcode = 4'b0110;

        // Test Case 1: sqrt(4.0) = 2.0

        a = 32'h40800000; // 4.0

        expected\_result = TWO\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 2: sqrt(1.0) = 1.0

        a = ONE\_POS;

        expected\_result = ONE\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 3: sqrt(0.0) = 0.0

        a = ZERO\_POS;

        expected\_result = ZERO\_POS;

        #10;

        test\_pass = (result == expected\_result);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 4: sqrt(-1.0) = NaN

        a = ONE\_NEG;

        #10;

        test\_pass = ((result[30:23] == 8'hFF) && (result[22] == 1)); // Check if result is NaN

        display\_result(a, b, result, opcode, test\_pass);

        // ----------------------------------------------------------------

        // Test 8: Equal (FEQ.S)

        // ----------------------------------------------------------------

        $display("\nTesting FEQ.S Instruction:");

        opcode = 4'b1000;

        // Test Case 1: 1.0 == 1.0 -> true (1)

        a = ONE\_POS;

        b = ONE\_POS;

        expected\_result = 32'h00000001;

        #10;

        test\_pass = (result[0] == expected\_result[0]);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 2: 1.0 == 2.0 -> false (0)

        a = ONE\_POS;

        b = TWO\_POS;

        expected\_result = 32'h00000000;

        #10;

        test\_pass = (result[0] == expected\_result[0]);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 3: +0.0 == -0.0 -> true (1)

        a = ZERO\_POS;

        b = ZERO\_NEG;

        expected\_result = 32'h00000001;

        #10;

        test\_pass = (result[0] == expected\_result[0]);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 4: NaN == 1.0 -> false (0)

        a = NAN\_CANONICAL;

        b = ONE\_POS;

        expected\_result = 32'h00000000;

        #10;

        test\_pass = (result[0] == expected\_result[0]);

        display\_result(a, b, result, opcode, test\_pass);

        // ----------------------------------------------------------------

        // Test 9: Less Than (FLT.S)

        // ----------------------------------------------------------------

        $display("\nTesting FLT.S Instruction:");

        opcode = 4'b1001;

        // Test Case 1: 1.0 < 2.0 -> true (1)

        a = ONE\_POS;

        b = TWO\_POS;

        expected\_result = 32'h00000001;

        #10;

        test\_pass = (result[0] == expected\_result[0]);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 2: 2.0 < 1.0 -> false (0)

        a = TWO\_POS;

        b = ONE\_POS;

        expected\_result = 32'h00000000;

        #10;

        test\_pass = (result[0] == expected\_result[0]);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 3: -1.0 < 1.0 -> true (1)

        a = ONE\_NEG;

        b = ONE\_POS;

        expected\_result = 32'h00000001;

        #10;

        test\_pass = (result[0] == expected\_result[0]);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 4: NaN < 1.0 -> false (0)

        a = NAN\_CANONICAL;

        b = ONE\_POS;

        expected\_result = 32'h00000000;

        #10;

        test\_pass = (result[0] == expected\_result[0]);

        display\_result(a, b, result, opcode, test\_pass);

        // ----------------------------------------------------------------

        // Test 10: Less Than or Equal (FLE.S)

        // ----------------------------------------------------------------

        $display("\nTesting FLE.S Instruction:");

        opcode = 4'b1010;

        // Test Case 1: 1.0 <= 2.0 -> true (1)

        a = ONE\_POS;

        b = TWO\_POS;

        expected\_result = 32'h00000001;

        #10;

        test\_pass = (result[0] == expected\_result[0]);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 2: 1.0 <= 1.0 -> true (1)

        a = ONE\_POS;

        b = ONE\_POS;

        expected\_result = 32'h00000001;

        #10;

        test\_pass = (result[0] == expected\_result[0]);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 3: 2.0 <= 1.0 -> false (0)

        a = TWO\_POS;

        b = ONE\_POS;

        expected\_result = 32'h00000000;

        #10;

        test\_pass = (result[0] == expected\_result[0]);

        display\_result(a, b, result, opcode, test\_pass);

        // Test Case 4: -1.0 <= 1.0 -> true (1)

        a = ONE\_NEG;

        b = ONE\_POS;

        expected\_result = 32'h00000001;

        #10;

        test\_pass = (result[0] == expected\_result[0]);

        display\_result(a, b, result, opcode, test\_pass);

        // ----------------------------------------------------------------

        // Test 11: Load and Store Operations

        // ----------------------------------------------------------------

        $display("\nTesting Load/Store Instructions:");

        // Test Load Operation (FLW)

        opcode = 4'b0000;  // Doesn't matter for load/store

        load = 1;

        store = 0;

        address = 32'h00000004;  // Aligned address

        mem\_data\_in = PI;  // Load value of PI

        #10;

        test\_pass = (result == PI);

        $display("Test FLW: mem[%h] = %h (%f), result = %h (%f) - %s",

                 address, mem\_data\_in, ieee\_to\_real(mem\_data\_in),

                 result, ieee\_to\_real(result),

                 test\_pass ? "PASS" : "FAIL");

        // Test Load Operation with misaligned address

        address = 32'h00000001;  // Misaligned address

        #10;

        $display("Test FLW (misaligned): address = %h, fcsr[3] should be set", address);

        // Test Store Operation (FSW)

        load = 0;

        store = 1;

        address = 32'h00000008;  // Aligned address

        a = E;  // Store value of E

        #10;

        test\_pass = (mem\_data\_out == E);

        $display("Test FSW: a = %h (%f), mem\_data\_out = %h (%f) - %s",

                 a, ieee\_to\_real(a),

                 mem\_data\_out, ieee\_to\_real(mem\_data\_out),

                 test\_pass ? "PASS" : "FAIL");

        // Test Store Operation with misaligned address

        address = 32'h00000002;  // Misaligned address

        #10;

        $display("Test FSW (misaligned): address = %h, fcsr[3] should be set", address);

        // Reset load/store signals

        load = 0;

        store = 0;

        // Test Register Write

        opcode = 4'b0000;  // ADD

        a = ONE\_POS;

        b = TWO\_POS;

        rd = 5'd10;  // f10 register

        write\_enable = 0;

        #10;

        // Trigger register write

        write\_enable = 1;

        #10;

        write\_enable = 0;

        $display("Test Register Write: Writing %h to f%0d", result, rd);

        // End simulation

        $display("\n--- All FPU Instructions Tested ---\n");

        #10 $finish;

    end

endmodule

`timescale 1ns / 1ps

module fpu\_tb;

    // Testbench signals

    reg [31:0] a, b;

    reg [31:0] mem\_data\_in;

    reg [3:0] opcode;

    reg load, store;

    reg [31:0] address;

    reg [4:0] rd;

    reg write\_enable;

    wire [31:0] result;

    wire [31:0] mem\_data\_out;

    // Instantiate the FPU module

    fpu uut (

        .a(a),

        .b(b),

        .mem\_data\_in(mem\_data\_in),

        .opcode(opcode),

        .load(load),

        .store(store),

        .address(address),

        .rd(rd),

        .write\_enable(write\_enable),

        .result(result),

        .mem\_data\_out(mem\_data\_out)

    );

    // Dump waveforms

    initial begin

        $dumpfile("fpu\_tb.vcd");

        $dumpvars(0, fpu\_tb);

    end

    // Helper tasks for floating-point operations

    task test\_binary\_op;

        input [3:0] op;

        input [31:0] operand\_a;

        input [31:0] operand\_b;

        input [4:0] dest\_reg;

        input [31:0] expected\_result;

        begin

            a = operand\_a;

            b = operand\_b;

            opcode = op;

            rd = dest\_reg;

            load = 0;

            store = 0;

            address = 0;

            write\_enable = 0;

            #5;

            write\_enable = 1;

            #5;

            write\_enable = 0;

            $display("Op: %b, A: %h, B: %h, Result: %h, Expected: %h, %s",

                     op, operand\_a, operand\_b, result, expected\_result,

                     (result === expected\_result) ? "PASS" : "FAIL");

            #10;

        end

    endtask

    task test\_unary\_op;

        input [3:0] op;

        input [31:0] operand\_a;

        input [4:0] dest\_reg;

        input [31:0] expected\_result;

        begin

            a = operand\_a;

            b = 32'h0;

            opcode = op;

            rd = dest\_reg;

            load = 0;

            store = 0;

            address = 0;

            write\_enable = 0;

            #5;

            write\_enable = 1;

            #5;

            write\_enable = 0;

            $display("Op: %b, A: %h, Result: %h, Expected: %h, %s",

                     op, operand\_a, result, expected\_result,

                     (result === expected\_result) ? "PASS" : "FAIL");

            #10;

        end

    endtask

    task test\_load\_store;

        input is\_load;

        input [31:0] operand\_a;

        input [31:0] mem\_address;

        input [31:0] memory\_data;

        input [4:0] dest\_reg;

        input [31:0] expected\_result;

        begin

            a = operand\_a;

            address = mem\_address;

            mem\_data\_in = memory\_data;

            rd = dest\_reg;

            load = is\_load;

            store = !is\_load;

            write\_enable = 0;

            #5;

            if (is\_load) begin

                write\_enable = 1;

                #5;

                write\_enable = 0;

                $display("Load: Addr: %h, Data: %h, Result: %h, Expected: %h, %s",

                         mem\_address, memory\_data, result, expected\_result,

                         (result === expected\_result) ? "PASS" : "FAIL");

            end else begin

                #5;

                $display("Store: Addr: %h, Data Out: %h, Expected: %h, %s",

                         mem\_address, mem\_data\_out, expected\_result,

                         (mem\_data\_out === expected\_result) ? "PASS" : "FAIL");

            end

            #10;

        end

    endtask

    // Main test sequence

    initial begin

        $display("Starting FPU Testbench");

        // Initialize

        a = 0;

        b = 0;

        mem\_data\_in = 0;

        opcode = 0;

        load = 0;

        store = 0;

        address = 0;

        rd = 0;

        write\_enable = 0;

        #10;

        $display("\n=== Testing FADD.S (ADD) ===");

        // 3.5 + 2.25 = 5.75

        test\_binary\_op(4'b0000, 32'h40600000, 32'h40100000, 5'd1, 32'h40B80000);

        // -2.5 + 1.5 = -1.0

        test\_binary\_op(4'b0000, 32'hC0200000, 32'h3FC00000, 5'd2, 32'hBF800000);

        $display("\n=== Testing FSUB.S (SUB) ===");

        // 3.5 - 2.25 = 1.25

        test\_binary\_op(4'b0001, 32'h40600000, 32'h40100000, 5'd3, 32'h3FA00000);

        // -2.5 - 1.5 = -4.0

        test\_binary\_op(4'b0001, 32'hC0200000, 32'h3FC00000, 5'd4, 32'hC0800000);

        $display("\n=== Testing FMUL.S (MUL) ===");

        // 3.5 \* 2.0 = 7.0

        test\_binary\_op(4'b0010, 32'h40600000, 32'h40000000, 5'd5, 32'h40E00000);

        // -2.5 \* 2.0 = -5.0

        test\_binary\_op(4'b0010, 32'hC0200000, 32'h40000000, 5'd6, 32'hC0A00000);

        $display("\n=== Testing FDIV.S (DIV) ===");

        // 3.5 / 2.0 = 1.75

        test\_binary\_op(4'b0011, 32'h40600000, 32'h40000000, 5'd7, 32'h3FE00000);

        // -8.0 / 2.0 = -4.0

        test\_binary\_op(4'b0011, 32'hC1000000, 32'h40000000, 5'd8, 32'hC0800000);

        $display("\n=== Testing FMIN.S ===");

        // min(3.5, 2.0) = 2.0

        test\_binary\_op(4'b0100, 32'h40600000, 32'h40000000, 5'd9, 32'h40000000);

        // min(-2.5, 1.5) = -2.5

        test\_binary\_op(4'b0100, 32'hC0200000, 32'h3FC00000, 5'd10, 32'hC0200000);

        $display("\n=== Testing FMAX.S ===");

        // max(3.5, 2.0) = 3.5

        test\_binary\_op(4'b0101, 32'h40600000, 32'h40000000, 5'd11, 32'h40600000);

        // max(-2.5, 1.5) = 1.5

        test\_binary\_op(4'b0101, 32'hC0200000, 32'h3FC00000, 5'd12, 32'h3FC00000);

        $display("\n=== Testing FSQRT.S ===");

        // sqrt(4.0) = 2.0

        test\_unary\_op(4'b0110, 32'h40800000, 5'd13, 32'h40000000);

        // sqrt(9.0) = 3.0

        test\_unary\_op(4'b0110, 32'h41100000, 5'd14, 32'h40400000);

        $display("\n=== Testing FEQ.S ===");

        // 3.5 == 3.5 ? 1 : 0 (true)

        test\_binary\_op(4'b1000, 32'h40600000, 32'h40600000, 5'd15, 32'h00000001);

        // 3.5 == 2.0 ? 1 : 0 (false)

        test\_binary\_op(4'b1000, 32'h40600000, 32'h40000000, 5'd16, 32'h00000000);

        $display("\n=== Testing FLT.S ===");

        // 2.0 < 3.5 ? 1 : 0 (true)

        test\_binary\_op(4'b1001, 32'h40000000, 32'h40600000, 5'd17, 32'h00000001);

        // 3.5 < 2.0 ? 1 : 0 (false)

        test\_binary\_op(4'b1001, 32'h40600000, 32'h40000000, 5'd18, 32'h00000000);

        $display("\n=== Testing FLE.S ===");

        // 2.0 <= 3.5 ? 1 : 0 (true)

        test\_binary\_op(4'b1010, 32'h40000000, 32'h40600000, 5'd19, 32'h00000001);

        // 3.5 <= 3.5 ? 1 : 0 (true)

        test\_binary\_op(4'b1010, 32'h40600000, 32'h40600000, 5'd20, 32'h00000001);

        $display("\n=== Testing FCVT.W.S (float to signed int) ===");

        // Convert 3.75 to int = 3

        test\_unary\_op(4'b1100, 32'h40700000, 5'd21, 32'h00000003);

        // Convert -2.25 to int = -2

        test\_unary\_op(4'b1100, 32'hC0100000, 5'd22, 32'hFFFFFFFE);

        $display("\n=== Testing FCVT.WU.S (float to unsigned int) ===");

        // Convert 3.75 to uint = 3

        test\_unary\_op(4'b1101, 32'h40700000, 5'd23, 32'h00000003);

        // Convert -2.25 to uint = 0 (negative becomes 0)

        test\_unary\_op(4'b1101, 32'hC0100000, 5'd24, 32'h00000000);

        $display("\n=== Testing FCVT.S.W (signed int to float) ===");

        // Convert int 42 to float = 42.0

        test\_unary\_op(4'b1110, 32'h0000002A, 5'd25, 32'h42280000);

        // Convert int -15 to float = -15.0

        test\_unary\_op(4'b1110, 32'hFFFFFFF1, 5'd26, 32'hC1700000);

        $display("\n=== Testing FCVT.S.WU (unsigned int to float) ===");

        // Convert uint 42 to float = 42.0

        test\_unary\_op(4'b1111, 32'h0000002A, 5'd27, 32'h42280000);

        // Convert uint 3000000000 (large value) to float

        test\_unary\_op(4'b1111, 32'hB2D05E00, 5'd28, 32'h4E32D05E);

        $display("\n=== Testing FLW (Load Float) ===");

        // Load float 3.5 from memory

        test\_load\_store(1, 32'h00000000, 32'h00000100, 32'h40600000, 5'd29, 32'h40600000);

        $display("\n=== Testing FSW (Store Float) ===");

        // Store float -2.5 to memory

        test\_load\_store(0, 32'hC0200000, 32'h00000200, 32'h00000000, 5'd0, 32'hC0200000);

        $display("\nFPU Testbench Completed");

        #50 $finish;

    end

endmodule