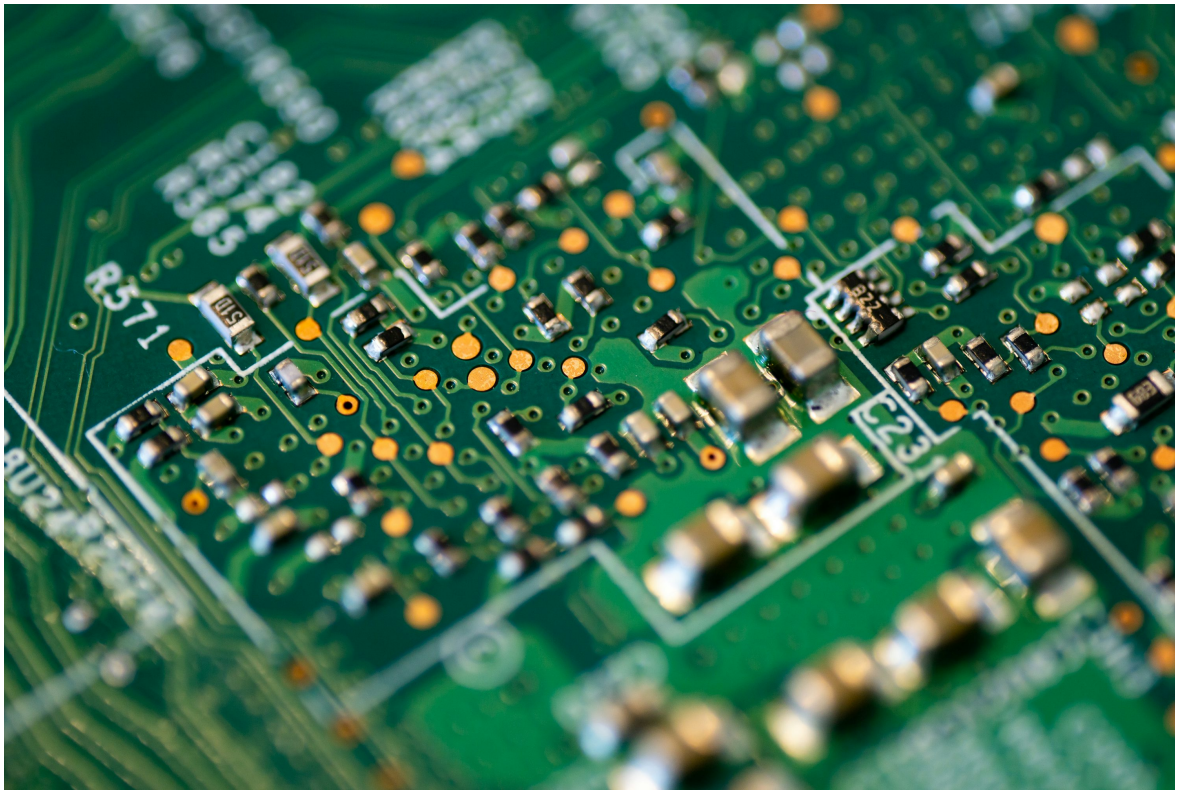
Digital HW Systems at Low Logic Levels II

Compulsory Coursework - Floating Point Multiplier



ΠΑΠΑΔΟΠΟΥΛΟΣ ΠΑΝΑΓΙΩΤΗΣ 10697

Academic Year 2024-2025

Contents

1 Introduction

This report presents the design, implementation, and verification of a pipelined single-precision floating-point multiplier, developed as part of the compulsory coursework for the course Digital HW Systems at Low Logic Levels II, during the academic year 2024–2025. The primary objective was to construct a hardware circuit in SystemVerilog that accurately performs floating-point multiplication in accordance with a modified IEEE 754 specification, and to validate its correctness through extensive simulation and formal verification techniques.

The floating-point multiplier is structured into four distinct modules: the main datapath unit (fp_mult), a normalization stage (normalize_mult), a rounding unit (round_mult), and an exception handling module (exception_mult). These components are integrated within a provided wrapper (fp_mult_top) that serves as the top-level module for simulation and verification. A key feature of the implementation is the incorporation of a pipeline register between the normalization and rounding stages, which enhances throughput by allowing partial overlap between successive operations.

Beyond functional implementation, the coursework emphasizes correctness through two complementary approaches: randomized and directed testing via a testbench, and formal validation using SystemVerilog Assertions (SVA). Random tests ensure general correctness across a wide input space, while corner case combinations verify correct behavior in edge conditions, including zeroes, infinities, NaNs, and denormals. Assertions further reinforce correctness guarantees by verifying relationships between status signals and output representations over time.

This document details the development process of the multiplier, including design decisions, module functionality, deviation from standard IEEE 754 behavior, and the methodology used to confirm correctness. The report also includes simulation results and assertion outcomes that confirm functional compliance with the coursework specification.

2 Design Overview

The floating-point multiplier is designed to operate on 32-bit single-precision floating-point inputs and to produce a 32-bit output in compliance with a modified IEEE 754 specification. The architecture of the multiplier is modular, with clearly defined boundaries between functional stages, promoting clarity, reusability, and ease of verification. The core datapath logic is implemented within the fp_mult module, while the fp_mult_top wrapper facilitates integration with the testbench and verification environment.

The design follows a pipelined structure composed of four primary modules:

1. **fp_mult** – The main datapath unit that orchestrates the entire multiplication operation across eight logical stages.
2. **normalize_mult** – Responsible for normalizing the product of mantissas and computing the associated exponent adjustments and rounding metadata.
3. **round_mult** – Implements rounding behavior based on the configured rounding mode and determines whether the result is inexact.
4. **exception_mult** – Detects and resolves corner cases such as multiplication involving zeroes, infinities, or NaNs, and generates the appropriate output and status signals.

The eight sequential stages within the fp_mult module are:

1. **Sign Calculation** – Determines the sign of the result by XORing the sign bits of the operands.
2. **Exponent Addition** – Computes the preliminary exponent sum.
3. **Bias Subtraction** – Adjusts the exponent by subtracting the IEEE 754 bias (127).
4. **Mantissa Multiplication** – Multiplies the extended mantissas (including the implicit leading 1).

5. **Normalization** – Shifts the mantissa product and adjusts the exponent to maintain proper floating-point representation.
6. **Pipeline Stage** – Introduces a register between normalization and rounding stages to support pipelining.
7. **Rounding** – Applies one of six possible rounding modes to the normalized mantissa, potentially increasing its bit-width.
8. **Exception Handling** – Handles overflow, underflow, and special operand categories (e.g., NaN, Inf, Zero) to finalize the output and generate status flags.

Each stage is carefully implemented to maintain modularity and clear signal separation. The pipeline stage ensures that once normalization is completed, subsequent stages can proceed independently of the current operation, thereby improving overall throughput. The design explicitly treats denormalized inputs as zero, as specified in the coursework, and applies specific behavior for operations such as ±0 × ±∞, which always result in positive infinity regardless of signs.

The outputs of the multiplier are:

- **z**: the final 32-bit floating-point result, incorporating sign, exponent, and mantissa.
- **status**: an 8-bit signal where each bit encodes specific conditions such as zero result, overflow, inexact operation, etc.

This modular and pipelined architecture enables both efficient computation and straightforward verification. The subsequent sections will elaborate on the functionality and implementation of each module, along with the strategies used to verify correctness across all input conditions.

```
// Figure : Module hierarchy of the floating-point multiplier system

// TOP TESTBENCH
// └── fp_mult_tb.sv                    <-- Testbench that drives the
                                            system
//       └── fp_mult_top.sv              <-- Wrapper module for DUT
//            └── fp_mult.sv             <-- Main datapath module
//                 ├── normalize_mult.sv     <-- Normalization stage
//                 ├── round_mult.sv         <-- Rounding stage
//                 └── exception_mult.sv     <-- Exception handling stage

// [fp_mult_tb] → [fp_mult_top] → [fp_mult]
//                                 ├ normalize_mult
//                                 ├ round_mult
//                                 └ exception_mult
```

3 Module Descriptions

This section provides a detailed description of each functional component that composes the floating-point multiplier. The implementation is structured around modularity and pipelining to enhance clarity and performance. The central unit, fp_mult, orchestrates the multiplication process through all functional stages, while normalize_mult, round_mult, and exception_mult perform dedicated computations within that pipeline. The modules are instantiated hierarchically, with fp_mult_top acting as the wrapper that interfaces with the testbench.

3.1 Main Module

The fp_mult module is the core of the floating-point multiplier. It receives two 32-bit floating-point inputs (a and b) and a 3-bit rounding mode signal (rnd), and produces a 32-bit result (z) along with an

8-bit status signal. Internally, it orchestrates the eight key stages of the multiplication process, as outlined below:

1. **Sign Calculation:**

The output sign is computed as the XOR of the sign bits of the two operands.

2. **Exponent Addition:**

The biased exponents of a and b are extracted and summed. The implicit bias (127) is subtracted to form the correct exponent base for normalized floating-point representation.

3. **Bias Subtraction**

The result of the exponent addition is adjusted by subtracting the IEEE 754 bias value (127), yielding the preliminary unnormalized exponent.

4. **Mantissa Multiplication with Leading Ones:**

Each operand's mantissa is extended with an implicit leading 1 (unless it is a denormal, which is treated as zero). The 24-bit mantissas are then multiplied to produce a 48-bit product.

5. **Normalization:**

The 48-bit product and biased exponent are passed to the normalize_mult module, which aligns the leading one and adjusts the exponent accordingly. It also computes the guard and sticky bits needed for rounding.

6. **Pipelining:**

A pipeline register is inserted between the normalization and rounding stages. This stage captures the normalized mantissa, adjusted exponent, and rounding metadata on the rising clock edge, and initializes to zero on an active-low reset. This pipelining is explicitly required in the coursework.

7. **Rounding:**

The round_mult module applies one of six rounding modes using the normalized mantissa (including leading one), guard, sticky, and sign information. It outputs a 25-bit mantissa (to account for overflow) and an inexact flag. After rounding, a post-rounding normalization step checks the MSB of the 25-bit result. If this bit is 1, a right shift is applied to the mantissa and the exponent is incremented.

8. **Exception Handling:**

The exception_mult module finalizes the computation by handling overflow, underflow, and special operand categories. It generates the final z output and computes six condition flags (zero_f, inf_f, nan_f, tiny_f, huge_f, inexact_f) that are encoded into the status bus.

The final status output aggregates these condition flags into an 8-bit word (bits 6 and 7 are unused). Overflow is flagged if the post-rounding exponent exceeds 254, and underflow is flagged if it drops below 1, per single-precision rules.

3.2 Normalization Module

The normalize_mult module is responsible for aligning the binary point of the 48-bit product resulting from mantissa multiplication, ensuring that the output adheres to the normalized floating-point representation. It also provides auxiliary metadata (guard and sticky bits) required for the rounding stage.

The module receives as input the 48-bit signal P, which represents the product of the two mantissas with implicit leading ones, and a 10-bit signed exponent exp_in, which reflects the biased exponent after subtraction. Based on the magnitude and alignment of P, the module determines whether a shift of the binary point is required and adjusts the exponent accordingly.

If the most significant bit P[47] is set, this indicates that the number begins with binary 10 or 11, and normalization requires a left shift by one bit. Consequently, the exponent is incremented by one.

Otherwise, if the leading bits are 01 (i.e., P[47] = 0 and P[46] = 1), no shifting is needed, and the exponent remains unchanged.

The module extracts the normalized mantissa as a 23-bit value, omitting the implicit leading one. The bit immediately to the right of the mantissa is provided as the guard bit, while the sticky bit is computed as the logical OR of all remaining lesser significant bits. These two bits are crucial in determining rounding behavior in the subsequent stage.

The module produces three outputs: the 23-bit normalized mantissa mant_out, the adjusted 10-bit signed exponent exp_out, and the guard and sticky flags. All logic is purely combinational, and the outputs are passed directly into the pipeline stage of the main module.

3.3 Rounding Module

The round_mult module is responsible for rounding the normalized mantissa to a 24-bit or 25-bit value based on the rounding mode specified by the input round. This stage ensures that the final result adheres to the selected rounding behavior while preserving IEEE 754 compliance, adapted to the coursework's modifications.

The module receives as input a 24-bit mantissa composed of the leading one concatenated with the 23-bit normalized mantissa produced by the previous stage. It also receives the guard and sticky bits, which represent the immediate and extended remainder bits, respectively, as well as the sign bit of the result and the 3-bit rounding mode.

Based on the combination of the rounding mode, the sign, and the values of the guard and sticky bits, the module determines whether to increment the mantissa by one (i.e., rounding up) or leave it unchanged (i.e., rounding down). The rounding modes supported include IEEE_near, IEEE_zero, IEEE_pinf, IEEE_ninf, and two coursework-specific behaviors (away_zero and near_up). These are enumerated in the round_defs.sv file and interpreted via a SystemVerilog enum construct within the module.

The result is a 25-bit mantissa. This extended width accounts for possible overflow caused by rounding up, which is handled during post-rounding normalization. The module also produces a 1-bit flag, inexact, which indicates whether any nonzero information existed in the bits discarded during rounding. Specifically, inexact is asserted if either the guard or sticky bit is set, implying that the mantissa had to be approximated.

All computations in round_mult are purely combinational. The outputs are forwarded back to the main fp_mult module, where the post-rounding normalization is performed by checking the MSB of the 25-bit result. If the MSB is high, the mantissa is right-shifted by one and the exponent is incremented to maintain normalization. Otherwise, both remain unchanged.

The design of this module follows the IEEE 754 rounding guidelines but includes default behavior fallback to IEEE_near if an invalid rounding mode is provided, in accordance with the coursework specification.

3.4 Exception Handling Module

The exception_mult module is responsible for handling all special and corner cases that may arise during floating-point multiplication. It ensures that the final output adheres to the modified IEEE 754 behavior specified in the coursework, including the handling of overflows, underflows, zeroes, infinities, and NaNs. This module also produces a set of six flags that collectively describe the exceptional status of the result.

The module receives as input the original operands a and b, the intermediate post-rounding result z_calc, the selected rounding mode round, and three control signals: ovf (overflow), unf (underflow), and inexact. The rounded result z_calc is derived in the main module, and serves as a candidate output in the absence of exceptions. The rounding mode is used to determine how extreme values should be handled when overflow or underflow occurs.

At the heart of the module is a classification mechanism based on an enumerated type interp_t, which categorizes a given 32-bit floating-point value into one of several predefined groups: ZERO, INF, NORM, MIN_NORM, or MAX_NORM. Notably, denormalized numbers are interpreted as zero, and NaNs are treated as infinities, in accordance with the coursework rules.

To support this logic, the module defines two functions:

- num_interp, which classifies a 32-bit input value into an interp_t category.
- z_num, which maps an interp_t category back to its corresponding 31-bit unsigned representation (excluding the sign bit). This function is used to construct canonical output values in overflow and underflow cases.

Within an always_comb block, the module first initializes all output flags and the result to zero. It then applies a case-based analysis using the interpreted values of a and b, following the coursework's special case table. For combinations such as NaN × anything, Inf × Zero, or Zero × Inf, the module produces predefined results, overriding the calculated sign to match the rule that $0 \times \infty = +\infty$ regardless of operand signs.

For the normal × normal case, the module examines whether overflow or underflow has occurred:

- In overflow cases, the output is set either to positive/negative infinity or to the maximum normalized value, depending on the rounding mode and the sign of the result.
- In underflow cases, the output is rounded to either zero or the minimum normalized value, again depending on rounding mode and sign.
- If neither condition applies, the result z_calc is forwarded as the final output.

Finally, the module sets the status flags:

- zero_f is asserted when the output is zero.
- inf_f is set when the result is infinity.
- nan_f is activated if the inputs correspond to a signaling or quiet NaN.
- tiny_f and huge_f indicate underflow and overflow events, respectively.
- inexact_f reflects whether any rounding approximation was necessary.

These flags are forwarded to fp_mult, where they are assembled into the 8-bit status output.

The behavior of this module is shaped by several coursework-specific deviations from the IEEE 754 standard. Notably, all denormalized numbers are treated as zero; this affects both input interpretation and classification via num_interp. Additionally, the case of ± 0 multiplied by $\pm\infty$ is explicitly defined to yield positive infinity, overriding the signs of the operands. These rules are hardcoded in the case logic to ensure deterministic behavior. Furthermore, any invalid rounding mode defaults to IEEE_near, as enforced in the downstream rounding module. The module also simplifies the handling of NaNs by interpreting them as infinities during classification. These deviations were implemented in strict accordance with the coursework specification and are essential to passing the corner case validations described later in the report.

4 Testbench Design

The verification of the floating-point multiplier is performed using the SystemVerilog testbench fp_mult_tb.sv. The testbench evaluates the correctness of the design under both random and corner-case conditions, covering all six supported rounding modes. It also includes assertion-based verification and verbose logging options for detailed inspection.

The testbench operates on a 10 ns clock and includes a reset sequence to initialize the pipeline. The top-level module under test is fp_mult_top, which instantiates the full multiplier datapath. Inputs a, b, and rnd are driven directly, and outputs z and status are monitored.

Two assertion modules are bound to the design:

- **test_status_bits** checks that invalid combinations of status flags are never simultaneously asserted.
- **test_status_z_combinations** ensures that the value of z is consistent with the meaning of the status bits.

Both are activated after three pipeline stages using the enable_assertions flag, and can optionally print successful passes using the verbose_pass flag.

Two main test phases are controlled using boolean flags:

- **run_random:** when enabled, the testbench generates 500 (editable) random pairs of operands for each of the six rounding modes. Results are compared against a reference result produced by the multiplication function, which takes a string-formatted rounding mode.
- **run_corner:** when enabled, the testbench evaluates all 144 combinations (12 × 12) of predefined corner-case values, including various forms of NaN, ±Inf, ±Zero, denormals, and normal values. Each case is tested under every rounding mode.

A verbose flag controls whether matching outputs are printed to the console in addition to mismatches, which are always displayed. Pipeline latency is accounted for by waiting three cycles after inputs are applied before checking the outputs.

In addition to the automated test phases, a single manual test case is applied at the end of the simulation. This test confirms that the system handles a known floating-point multiplication correctly, and prints both the hexadecimal and real-number representations of inputs and outputs.

The testbench also generates a dump.vcd waveform file using *$dumpfile* and *$dumpvars* to support visual inspection of simulation traces.

Together, this setup ensures functional correctness, compliance with custom exception behavior, and assertion validity across both typical and pathological input spaces.

5 Corner Case Coverage

To verify the robustness of the multiplier under exceptional input conditions, the testbench includes a dedicated corner case validation phase. This phase exhaustively evaluates all meaningful combinations of floating-point edge values, in alignment with the coursework specifications.

A predefined set of 12 representative 32-bit floating-point patterns is constructed. These include:

- Two signaling NaNs (positive and negative)
- Two quiet NaNs (positive and negative)
- Two infinities (+∞, −∞)
- Two normal values (+1.0, −1.0)

- Two denormals (positive and negative)
- Two zeros (positive and negative)

Each of these values is used as operand a and operand b, forming a complete set of 12 × 12 = 144 unique input pairs. For each pair, the multiplication is computed under all six rounding modes, leading to a total of 864 test cases specifically targeting corner conditions.

The expected result is generated using the reference multiplication function, and the output of the multiplier is compared after accounting for the 3-cycle pipeline latency. Both mismatches and correct matches (if verbose is enabled) are logged with case identifiers and hexadecimal representations of the inputs and results.

This systematic strategy guarantees coverage for all combinations involving special categories such as:

- NaN × anything
- ±0 × ±∞
- denormal × normal
- ±∞ × normal, and others.

Special care is taken to validate the coursework-specific behavior, including the rule that any combination of ±0 and ±∞ yields +∞, regardless of signs. Additionally, assertion modules remain active during corner case testing, ensuring that status flags (e.g., inf_f, nan_f, tiny_f) remain mutually consistent and logically sound in all edge cases.

Through this exhaustive enumeration, the design is validated not only for numerical accuracy but also for correctness in status signaling and exception handling across the full range of IEEE 754 special cases as modified by the coursework.

6 Assertion-Based Verification

To ensure the internal correctness and mutual consistency of the output status signals and result, the design incorporates a formal assertion-based verification framework. Two dedicated SystemVerilog assertion modules, **test_status_bits** and **test_status_z_combinations**, are bound to the top-level module fp_mult_top during simulation. Assertions are enabled after three clock cycles using the enable_assertions flag, ensuring alignment with the internal pipeline latency.

The **test_status_bits** module focuses on the mutual exclusivity of status flags. It includes eight immediate assertions that validate the correctness of the following combinations:

- zero_f must never be high simultaneously with inf_f, nan_f, or huge_f.
- inf_f must not be high together with nan_f or tiny_f.
- nan_f must not overlap with tiny_f or huge_f.
- tiny_f and huge_f must never assert concurrently.

These checks prevent logically invalid or contradictory outputs. When the verbose_pass flag is enabled, successful pass messages are printed for all satisfied conditions.

The **test_status_z_combinations** module contains five concurrent assertions that verify consistency between the z output and the active status flags:

1. If zero_f is asserted, the exponent field of z must be zero.
2. If inf_f is asserted, the exponent field of z must be all ones (255).

3. If nan_f is asserted, three cycles earlier one operand must have had an exponent of 0 and the other 255, representing the $0 \times \infty$ condition.
4. If huge_f is asserted, z must either represent positive/negative infinity or the maximum normalized value.
5. If tiny_f is asserted, z must represent zero or the minimum normalized number.

For properties involving temporal dependencies (such as the nan_f condition), pipelining is implemented within the module using internal shift registers to store delayed versions of operands a and b. This allows assertions to inspect the state of inputs from three clock cycles in the past, aligning with the pipeline structure of the multiplier.

Assertions are conditionally enabled using the enable_assertions flag, and their pass messages are controlled via verbose_pass. All assertion failures generate descriptive errors with simulation timestamps, allowing for rapid identification of inconsistencies during waveform inspection or log analysis.

Together, these modules provide both logical and temporal coverage of status behavior, ensuring that exception flags are not only activated under the correct conditions but also remain internally coherent and mutually exclusive.

7 Simulation Results

The simulation process validated both the functional correctness and the exception handling behavior of the floating-point multiplier across all input categories and rounding modes. Two classes of test scenarios were executed: randomized tests and exhaustive corner case evaluations.

During the random testing phase, 500 test cases were generated per rounding mode, for a total of 3000 random input combinations. Each test compared the multiplier's output (z) with the result of the reference multiplication function after accounting for the 3-cycle pipeline delay. Mismatches were printed to the console with full operand and rounding mode context. When the verbose flag was enabled, passing cases were also reported, confirming alignment between the DUT and the expected results.

In the corner case phase, all 144 pairwise combinations of 12 special floating-point values were tested across six rounding modes, resulting in 864 deterministic tests. These cases included signaling and quiet NaNs, infinities, denormalized numbers, and both positive and negative zero and one. This ensured full coverage of all combinations relevant to the exception handling logic.

Assertions were enabled during both phases. The test_status_bits module verified that mutually exclusive status flags never asserted simultaneously, while test_status_z_combinations validated the consistency between output flags and the value of z. All assertions passed in every tested case. When verbose_pass was activated, confirmation messages for each passing assertion were printed to the console.

Waveform dumping via *$dumpfile* and *$dumpvars* allowed for post-simulation analysis in GTKWave. Key simulation traces — including the propagation of pipeline data, assertion triggers, and rounding transitions — were inspected visually and confirmed to follow the expected behavior.

A final manual test case involving specific hexadecimal operands was executed to provide a human-readable example. The inputs and outputs were displayed both in hexadecimal and floating-point representations, verifying the multiplier's correctness in handling extremely small magnitude values.

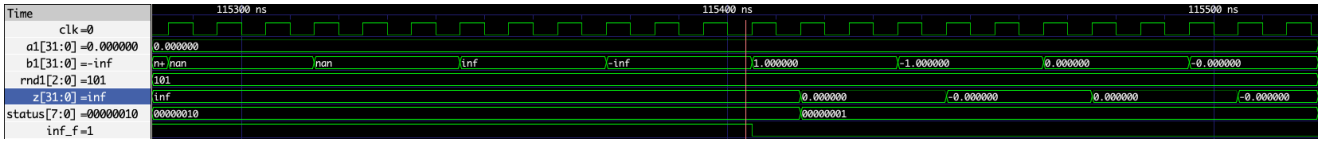
The combined simulation evidence demonstrates that the multiplier performs accurate floating-point multiplication, applies the correct rounding behavior, and fully conforms to the modified IEEE 754 requirements under both standard and exceptional conditions.

7.1 Simulation Setup

The simulation was conducted using Cadence Xcelium 23.09 on EDA Playground, as recommended for SystemVerilog-based designs with assertion support. The testbench (fp_mult_tb.sv) was compiled alongside the full design hierarchy, including fp_mult_top, fp_mult, and all submodules.

Assertions were not visible in the waveform output (dump.vcd), as per coursework instructions. Instead, they were printed directly to the simulator console using *$display()* and *$error()* statements, allowing real-time validation of status flags and data conditions.

The simulation was configured to produce a full waveform dump (dump.vcd), which includes all internal signals relevant to the design's functional stages, and is included in the submission package for reference.



Demonstration of correct 3-stage pipelined behavior in the floating-point multiplier. Inputs a = +0.0, b = −∞, and round = away_zero are applied at cycle T0. The output z = +∞ and inf_f = 1 status flag appear exactly three cycles later, following normalization, rounding, and exception handling stages. This behavior confirms correct pipeline implementation and alignment with the coursework-specific rule for ±0 × ±∞ = +∞.

8 Conclusion

This report presented the complete design, implementation, and verification of a pipelined single-precision floating-point multiplier developed in SystemVerilog. The architecture was constructed around modular units for normalization, rounding, and exception handling, with a central datapath (fp_mult) coordinating the full computation flow. A single pipeline stage between normalization and rounding was inserted, as specified in the coursework, enabling three-stage latency and improved throughput.

The design conforms to a modified IEEE 754 specification, including the treatment of denormalized numbers as zero and the mandated behavior for special cases such as ±0 × ±∞, which must always yield positive infinity. These behaviors were systematically implemented and validated through simulation.

Verification was performed using both randomized and exhaustive corner case testing. Over 3000 random test cases and 864 corner case combinations were applied under all six supported rounding modes. All outputs were compared against a trusted reference model, and status flags were thoroughly validated.

Assertion-based verification was employed using two dedicated SystemVerilog modules. These modules ensured the logical consistency of status flags and enforced relationships between flagged exceptions and result encoding. Assertions passed in all tested cases, confirming both the correctness and robustness of the implementation.

Simulation was executed using Cadence Xcelium 23.09 on EDA Playground, with waveforms captured in a full dump.vcd file. Assertion results were printed to the simulator console, as required by the coursework.

In conclusion, the multiplier operates correctly under all tested scenarios, fulfills all coursework specifications including special-case handling, and demonstrates a complete and verified pipelined floating-point unit.