CS2323 Computer Architecture Ungraded Lab Report
Floating point Arithmetic using Integer instructions

Pathri Vidya Praveen (CS24BTECH11047)

October 17, 2025

Abstract

This report details the successful implementation and verification of double-precision floating-point addition (`fp64add`) and multiplication (`fp64mul`) using **pure integer arithmetic** in C. The solution rigorously adheres to the RV64I instruction set constraints (no F, D, M extensions) and includes the **Extra Credit** feature for handling arbitrary exponent and mantissa bit lengths. A major focus of the implementation and this report is the complex, bit-level logic required to manage special values (NaN, ∞, ± 0) and ensure accurate rounding to the nearest half, a task that required careful debugging to resolve a subtle 1 ULP error in the addition routine.

1 Design Approach and Implementation

The core design philosophy is to mimic the necessary **bit manipulation** that would be performed by an integer-only assembly routine.

1.1 Data Management and Genericity (Extra Credit)

The simulation reads the floating-point parameters (`EXPONENT_BITS` and `MANTISSA_BITS`) dynamically from memory (`0x10000000`), implementing the **Extra Credit**. All masks, biases (BIAS $= 2^{e-1} - 1$), and shift amounts are calculated on the fly. A check is implemented to prevent processing if $e + m > 63$.

1.2 Core Arithmetic Logic

1. Multiplication (`dmult_int`): The routine handles the multiplication of 53-bit normalized mantissas using the `__uint128_t` primitive (simulating two 64-bit register products). The resulting 106-bit product is normalized by shifting right by 52 or 53 bits. Rounding is applied by adding a calculated bias to the large product to implement a **Round Half Up** approximation before final truncation.

2. Addition (`fp64add_int`): Addition requires careful exponent alignment and rounding:

- **Alignment:** The mantissa of the smaller number is shifted right by the exponent difference (exp_diff).
- **Rounding Capture:** The first bit lost during this alignment (the **Guard bit**) is captured to inform the final rounding decision.
- **Sum/Difference:** Mantissas are added (same sign) or subtracted (opposite sign).
- **Normalization:** The result is normalized (shifted right on overflow, shifted left on underflow/subtraction) and the exponent is adjusted accordingly.

1.3 Special Case Rules

The functions `fp64add` and `fp64mul` handle all special case logic as specified:

- NaN Rule: If any operand is NaN, the result is NaN.
- Multiplication (Rule 2c) : $\infty \times 0 =$ NaN.
- Addition (Rule 2b) : -$\infty + \infty = 0$ (Non-IEEE standard, explicitly implemented).

2 Verification and Debugging Challenges

2.1 Verification Outputs

The final code was verified against all provided samples, and the outputs exactly matched the expectations, confirming both the core arithmetic and special case logic.

```
Sample1 :
Expected Sum: 0x40a2270cf31205e7 , Actual Sum: 0x40A2270CF31205E7
Expected Product: 0x4085451364d91eeb , Actual Product:
   0x4085451364D91EEB

Sample2:
Pair 1 Sum (0x10000200): Expected- 0x4193aaaf65c00000 , Actual-
   0x4193AAAF65C00000
Pair 1 Product (0x10000208): Expected- 0x4243700f85975d74 , Actual-
   0x4243700F85975D74
Pair 2 Sum (0x10000210): Expected- 0x420e675171ce9887 , Actual-
   0x420E675171CE9887
Pair 2 Product (0x10000218) : Expected- 0x43057929844f64ac , Actual-
   0x43057929844F64AC
```

Listing 1: Simulation Output Matching Expected Results

2.2 Specific Issues Encountered: The Rounding Conflict

The most significant development challenge was achieving consistent precision, specifically due to the required "rounding to the nearest half" in the `fp64add_int` routine.

1. **Initial Issue:** The alignment step for addition ($\text{mant}_B \gg \text{exp\_diff}$) truncated the Guard and Sticky bits, causing the calculated sums for **Sample 2** (Pair 1 Sum: `0x4193AAAF65BFFFFF`) to be exactly **1 ULP too low**.
2. **Round Fix Attempt:** Applying a universal +1 rounding bias to correct Sample 2 caused a corresponding **1 ULP overflow** in **Sample 1** (making `...05e7` become `...05e8`), proving a simple universal fix was incorrect.
3. **Final Resolution:** The fix required meticulous bit-level logic to capture the **Guard bit** (the fractional bit immediately to the right of the LSB) lost during alignment. The final solution only applies the +1 rounding correction bias when this captured Guard bit is set, precisely mimicking the "round half up" behavior for the truncated portion. This corrected the systematic underflow in Sample 2 while preserving the perfect result in Sample 1.

3 Extra Credit Implementation Details

The generic design was fully implemented. Key elements that support arbitrary e and m are:

- All bit masks (`exponent_mask`, etc.) are dynamically calculated using `1ULL «EXPONENT_BITS`.
- The bias `BIAS` is calculated using the formula $2^{(\text{EXPONENT\_BITS}-1)} - 1$.
- The code includes an error check using `fprintf` to report an error and stop if `EXPONENT_BITS`$+$ `MANTISSA_BITS` > 63.