BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP.HCM

KHOA ĐIỆN – ĐIỆN TỬ

BÁO CÁO CUỐI KỲ

THIẾT KẾ BỘ TÍNH TOÁN SỐ THỰC DẤU PHẨY ĐỘNG

MÔN HỌC: THIẾT KẾ HỆ THỐNG VÀ VI MẠCH TÍCH HỢP

LỚP HỌC PHẦN: ICSD336764_22_2_11

GVHD: TS ĐỖ DUY TÂN

SINH VIÊN THỰC HIỆN:

ĐỖ TRUNG HẬU	**- 21161121**
PHAN VĂN NGUYÊN	**- 21161160**
NGUYỄN QUỲNH ĐÌNH	**- 21161115**
PHẠM LÊ TRƯỜNG VŨ	**- 21161220**
NGUYỄN THỊ TRÚC MAI	**- 21161147**

Tp. Hồ Chí Minh - 5/2023

ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP.HCM
KHOA ĐIỆN - ĐIỆN TỬ
www.utex.hcmute.edu.vn

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM

Độc lập – Tự do – Hạnh phúc

Thủ Đức, Tháng 5, Năm 2023

DANH SÁCH THÀNH VIÊN THAM GIA VIẾT BÁO CÁO

HỌC KỲ 2 NĂM HỌC 2023 – 2024

Lớp học phần: ICSD336764_22_2_11

Giảng viên hướng dẫn: TS Đỗ Duy Tân

Tên đề tài: *"THIẾT KẾ BỘ TÍNH TOÁN SỐ THỰC DẤU PHẨY ĐỘNG".*

STT	Họ và tên sinh viên	Mã số sinh viên	Số điện thoại
1	Đỗ Trung Hậu	21161121	
2	Phan Văn Nguyên	21161160	
3	Nguyễn Quỳnh Đình	21161115	
4	Phạm Lê Trường Vũ	21161220	
5	Nguyễn Thị Trúc Mai	21161147	

Nhận xét của giáo viên

.....

.....

.....

.....

.....

.....

Tp. Hồ Chí Minh – Tháng 5 năm 2023

ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP.HCM
KHOA ĐIỆN - ĐIỆN TỬ
www.utex.hcmute.edu.vn

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM

Độc lập – Tự do – Hạnh phúc

Thủ Đức, Tháng 5, Năm 2023

Ý KIẾN CỦA GIÁO VIÊN HƯỚNG DẪN

Nhóm sinh viên:

Đỗ Trung Hậu	- 21161121
Phan Văn Nguyên	- 21161160
Nguyễn Quỳnh Đình	- 21161115
Phạm Lê Trường Vũ	- 21161220
Nguyễn Thị Trúc Mai	- 21161147

Ngành: Công nghệ kĩ thuật Điện tử - Viễn thông

Năm học: 2021-2022 ***Lớp:*** 211612C

***Đề tài*:** Thiết kế bộ tính toán số thực dấu phẩy động

Giảng viên: TS Đỗ Duy Tân

Ý kiến

1. Nội dung của đề tài:

- ....

2. Ưu điểm:

- ....
- ....

3. Nhược điểm:

-
.....

-
.....

4. Xếp loại:

- ....

5. Điểmsố:.....(Bằngchữ:).....

ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP.HCM
KHOA ĐIỆN - ĐIỆN TỬ
www.utex.hcmute.edu.vn

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM

Độc lập – Tự do – Hạnh phúc

Thủ Đức, Tháng 5, Năm 2023

LỜI CẢM ƠN

Để hoàn thành tốt bài báo cáo dự án cuối kì, ngoài sự nổ lực của bản thân thì chúng em còn nhận được sự quan tâm giúp đỡ của Thầy và mọi người xung quanh.

Đặc biệt chúng em xin gửi đến TS Đỗ Duy Tân - người đã tận tình hướng dẫn, giúp đỡ chúng em để hoàn thành bài báo cáo này một lời cảm ơn chân thành và sâu sắc nhất.

Chúng em thực sự biết ơn đến tất cả bạn bè của chúng tôi, những người đã có những góp ý quý báu trong thời gian chúng tôi làm dự án. Chúng em cũng phải thừa nhận các nguồn tài nguyên học tập mà chúng em đã nhận được từ nhiều nguồn khác nhau.

Vì thời gian nghiên cứu có hạn, trình độ hiểu biết của bản chúng em còn nhiều hạn chế nên bài báo cáo của chúng em không tránh khỏi những thiếu sót, em rất mong nhận được sự góp ý quý báu của tất cả các thầy cô giáo để báo cáo của chúng em được hoàn thiện hơn.

Chúng em xin chân thành cảm ơn.

Thủ Đức, Tháng 5, Năm 2023

Nhóm thực hiện

DANH SÁCH CÁC BẢNG

DANH SÁCH CÁC HÌNH ẢNH

MỤC LỤC

CHƯƠNG I: TỔNG QUAN

1.1 Đặt vấn đề

Khối số thực dấu phẩy động rất thông dụng trong bộ đồng xử lý toán học. Nó là một phần của một hệ thống máy tính được thiết kế đặc biệt để thực hiện các hoạt động tính toán trên các số thực dấu chấm động. Một số hoạt động tính toán trên khối FPU như: cộng, trừ, nhân, chia. Mục đích là để xây dựng một CPU hiệu quả để thực hiện các chức năng cơ bản cũng như chức năng siêu việt với việc làm giảm độ phức tạp của logic được sử dụng làm giảm hoặc ít nhất giới hạn thời gian tương đương như dòng x87 và làm giảm bộ nhớ càng nhiều càng tốt. Để thực hiện các chức năng như: cộng, trừ, nhân , chia…. các số thực dấu phẩy động phải chuyển đổi dữ liệu sang định dạng chuẩn IEEE 754. Tất cả thuật toán trên đã được đồng bộ hóa và đánh giá theo môi trường Spartan 3E Synthesis. Tất cả các chức năng được xây dựng bởi các thuật toán có hiệu quả với một số thay đổi kết hợp ở cuối trong phạm vi cho phép. Khi một CPU thực hiện một chương trình được gọi đến một số thực dấu phẩy động (FP) hoạt động, có ba cách nó có thể thực hiện. Thứ nhất, nó có thể gọi là một số thực dấu phẩy động giả lập, đó là một thư viện số thực dấu phẩy động, sử dụng một loạt các phép tính số học số thực dấu phẩy động được thực hiện trên khối ALU. Các giả lập có thể lưu trên các phần cứng bổ sung của một FPU nhưng chậm đáng kể. Thứ hai, nó có thể sử dụng một thêm một FPU được hoàn toàn tách biệt với CPU, nó chỉ cần thiết để tăng tốc độ tính toán . Còn lại là sử dụng tích hợp FPU có trong hệ thống. Đây chính là cái mà chúng ta quan tâm đến cũng như là thiết kế ra một phiên bản đơn giản nhất của nó, tất cả được trình bày chi tiết trong quyển báo cáo này.

1.2 Mục tiêu của đề tài

Mục tiêu của đề tài là xây dựng nên một bộ tính toán dành cho các số thực dấu phẩy động được biểu diễn theo chuẩn IEEE 754 – 32 bit với 4 phép toán cơ bản là: cộng, trừ, nhân và chia. Các phép toán được phân biệt với nhau bởi 2 bit lựa chọn, đồng thời xuất kết quả của phép tính ở ngõ ra và báo hiệu việc tràn số cũng như các trường hợp ngoại lệ khác.

1.3 Nội dung thực hiện

Để đạt được mục tiêu trên, đề tài phải đạt được những nội dung sau đây:

+ Trình bày lý thuyết khối FPU trong máy tính.

+ Tìm hiểu lý thuyết về phương pháp chuẩn hóa IEEE 754 trong tính toán số học số thực dấu phẩy
động.

+ Trình bày thuật toán cộng, trừ, nhân, chia số thực dấu phẩy động với 23 bít phần định trị và 8 bít phần số mũ. Tất cả số âm phần định trị và số mũ được thể hiện ở dạng nhị phân bù hai.

+ Viết được chương trình cho khối số học số thực dấu phẩy động bằng verilog HDL với các phép tính cộng, trừ, nhân, chia.

+ Mô phỏng kết quả đạt được thông qua file testbench.

+ Đánh giá, nhận xét và phát triển đề tài trong tương lai.

1.4 Bố cục của đề tài

Bài báo cáo được chia thành 6 chương chính:

+ **Chương 1:** Tổng quan

+ **Chương 2:** Cơ sở lý thuyết về biểu diễn số thực dấu phẩy động theo chuẩn IEEE

+ **Chương 3:** Các module phục vụ và sơ đồ khối của bộ FPU

+ **Chương 4:** Thiết kế và giải thuật cho các khối trong bộ FPU

+ **Chương 5:** Đánh giá kết quả đạt được qua testbench.

+ **Chương 6:** Kết luận và hướng phát triển

1.5 Giới hạn của đề tài

Do còn nhiều giới hạn về kiến thức cũng như thời gian tìm hiểu, chính vì vậy với đề tài: "*Thiết kế bộ tính toán số thực dấu phẩy động"* , nhóm chúng em tập trung nghiên cứu vào việc thiết kế và giải thuật cho các khối phép tính cộng, trừ, nhân, chia các số thực dấu phẩy động theo chuẩn IEEE 754 với số bit biểu diễn là 32 bit.

CHƯƠNG II: CƠ SỞ LÝ THUYẾT VỀ BIỂU DIỄN SỐ THỰC DẤU PHẨY ĐỘNG THEO CHUẨN IEEE

2.1 Giới thiệu về phương pháp biểu diễn số thực dấu phẩy động

2.1.1 Thuật ngữ dấu phẩy động

Trong tin học, dấu phẩy động được dùng để chỉ một hệ thống biểu diễn số mà trong đó sử dụng một chuỗi chữ số (hay bit) để biểu diễn một số hữu tỉ.

Thuật ngữ *dấu phẩy động* xuất phát từ chỗ hệ thống dấu phẩy động có dấu phẩy cơ số (tức là dấu phẩy thập phân trong trường hợp dùng hệ thập phân thường ngày hoặc là dấu phẩy nhị phân trong trường hợp dùng bên trong máy tính) không cố định mà có thể thay đổi vị trí của nó bất kỳ đâu trong các chữ số có nghĩa của số cần được biểu diễn. Vị trí này được mô tả một cách độc lập trong biểu diễn cụ thể của từng số. Đã có nhiều hệ thống dấu phẩy động khác nhau được dùng trong máy tính; tuy nhiên, vào khoảng hai mươi năm trở lại đây thì hầu hết các máy tính đều dùng cách biểu diễn tuân thủ theo chuẩn IEEE 754.

Một điều cần lưu ý là có sự khác biệt khi gọi tên dấu phẩy cơ số: ở Việt Nam, chúng ta dùng dấu phẩy để ngăn cách giữa phần nguyên và phần thập phân; trong khi các nước như Mỹ, Anh,…dùng dấu chấm để làm điều này. Chính vì thế, thuật ngữ tương ứng với dấu phẩy động ở tiếng Anh là floating point mà dịch sát ra tiếng Việt phải là dấu chấm động. Các phần sau, ta vẫn dùng thuật ngữ dấu phẩy động nhưng khi biểu diễn các số trong ví dụ thì vẫn mô tả theo quy ước của các nước nói trên (mà cũng là quy ước chuẩn dùng trong máy tính), nghĩa là vẫn dùng dấu chấm thay cho dấu phẩy.

Ưu điểm của cách biểu diễn dấu phẩy động là nó cho phép biểu diễn một tầm giá trị rộng hơn nhiều so với cách biểu diễn dấu phẩy tĩnh. Lấy ví dụ, nếu cách biểu diễn dấu phẩy tĩnh có bảy chữ số thập phân với quy ước dấu phẩy thập phân luôn nằm cố định ở chữ số thứ năm, thì cách biểu diễn dấu phẩy tĩnh có thể mô tả các số như 12345.67, 8765.43, 123.00 (tức 00123.00) và vân vân. Trong khi đó cách biểu diễn dấu phẩy động (chẳng hạn như định dạng decimal32 của IEEE 754) với bảy chữ số thập phân ngoài việc mô tả được các số nói trên còn mô tả được nhiều số khác mà dấu phẩy tĩnh không mô tả được như 1.234567, 123456.7, 0.00001234567, 1234567000000000, và nhiều nữa.

Tất nhiên, định dạng theo kiểu dấu phẩy động cần thêm bộ nhớ hơn so với dấu phẩy tĩnh (vì cần có thêm bộ nhớ để mô tả vị trí của dấu phẩy cơ số), nhưng ta có thể nói: với cùng một không gian bộ nhớ, cách biểu diễn dấu phẩy động đạt được tầm mô tả rộng hơn.

2.1.2 Biểu diễn số thực dưới dạng dấu phẩy động

Một cách biểu diễn số (gọi là hệ thống ký số trong toán học) bao gồm phương pháp dùng để lưu trữ một con số bằng một chuỗi các chữ số. Số học được định nghĩa là tập hợp các thao tác cần làm trên cách biểu diễn số đã cho để thực hiện các phép toán số học (cộng, trừ, nhân, chia…).

Có một số cách dùng chuỗi các chữ số để biểu diễn các con số. Theo ký hiệu toán học thông dụng, chuỗi chữ số có thể có chiều dài tùy ý, và vị trí của dấu phẩy cơ số được chỉ ra bằng cách đặt một ký tự rõ ràng (đó là dấu chấm đối với các nước Anh, Mỹ… hoặc là dấu phẩy đối với Việt Nam). Trong trường hợp chuỗi chữ số không có dấu phẩy thì nó được xem như đặt ở phía cuối bên phải của chuỗi chữ số (tức là, số đang biểu diễn là một số nguyên). Trong trường hợp máy tính thì với chỉ hai bit 0 và 1, không thể có một ký tự rõ ràng phân biệt để mô tả dấu phẩy. Trong hệ thống dấu phẩy tĩnh, người ta quy ước vị trí cố định của dấu phẩy cơ số trong chuỗi chữ số. Lấy ví dụ, quy ước rằng chuỗi chữ số gồm 8 chữ số thập phân và dấu phẩy thập phân luôn nằm ở ngay giữa chuỗi thì khi đọc giá trị "00012345" ta phải ngầm hiểu đây là số có giá trị 1.2345.

Trong ký hiệu khoa học, một con số thường được lấy tỉ lệ (tức được nhân) với một lũy thừa của 10 sao cho kết quả sau khi lấy tỉ lệ nằm trong một tầm cho trước – điển hình là nằm trong khoảng 1 và 10, tức là kết quả sẽ được viết ra với dấu phẩy cơ số nằm trực tiếp sau chữ số đầu tiên. Để người đọc có thể biết giá trị thực của con số, lũy thừa của 10 sẽ được viết riêng ra ở cuối kết quả. Lấy ví dụ, chu kỳ xoay mặt trăng Io của hành tinh Mộc Tinh là 152853.5047 giây. Khi đó, con số này được biểu diễn dưới dạng ký hiệu khoa học chuẩn là 1.528535047×10^5 giây.

Cách biểu diễn số dấu phẩy động tương tự với cách dùng trong ký hiệu khoa học. Mô tả luận lý thì một số dấu phẩy động bao gồm:

+ Một chuỗi chữ số có dấu với chiều dài cho trước và có cơ số cho trước. Chuỗi này được gọi là phần định trị. Dấu phẩy cơ số không được thể hiện tường minh ở phần này, nhưng được quy ước ngầm là luôn luôn nằm tại một vị trí cụ thể trong phần định trị - mà thường là ngay sau hoặc ngay trước chữ số có nghĩa lớn nhất. Bài viết này nếu không nói rõ sẽ tuân theo quy ước là dấu phẩy cơ số luôn ở ngay sau chữ số có nghĩa lớn nhất (tức là chữ số đầu tiên tính từ bên trái qua). Độ dài của phần định trị xác định độ chính xác mà các con số có thể được biểu diễn.
+ Một số mũ là số nguyên có dấu, nhằm mô tả phần lấy tỉ lệ tức cho phép người đọc xác định được giá trị thực của số từ phần định trị.

Thử lấy một ví dụ ở cơ số 10 (hệ thập phân mà ta thường dùng hằng ngày) với số 152853.5047 mà có độ chính xác là mười chữ số thập phân. Khi biểu diễn dưới dạng dấu phẩy động thì nó sẽ được viết với phần định trị là 1528535047 (với quy ước là vị trí của dấu phẩy cơ số nằm ngay sau chữ số có nghĩa lớn nhất, tức là chữ số 1). Khi đó, phần định trị được hiểu ngầm là 1.528535047. Để người đọc có thể khôi phục lại giá trị ban đầu thì cần phải thêm số mũ là 5. Khi đó, người đọc sẽ nhân giá trị của phần định trị (sau khi đã thêm dấu phẩy cơ số vào vị trí quy ước) với 105 để được 1.528535047×10^5, hay 152853.5047.

Mô tả hình thức thì giá trị cuối cùng của một số dấu phẩy động là: $s\times b^e$, với s là giá trị của phần định trị (sau khi đã đặt dấu phẩy cơ số vào vị trí quy ước), b là cơ số, và e là số mũ.

Hoàn toàn tương đương, có thể viết công thức trên như sau: $\frac{S}{b^{p-1}} \times b^e$, với S là giá trị nguyên của toàn bộ phần định trị mà chưa đặt dấu phẩy cơ số và p là độ chính xác – số chữ số của phần định trị.

Khi dùng cách biểu diễn dấu phẩy động, phương pháp lưu trữ phần định trị, số mũ và bit dấu bên trong máy tính tùy thuộc vào mỗi loại máy theo chuẩn nào. Hiện nay, chuẩn IEEE 754 là thông dụng nhất sẽ được mô tả ở phần sau. Nhưng ở đây, chúng ta thử xét định dạng nhị phân độ chính xác đơn (32 bit) của IEEE754. Theo chuẩn này thì một số dấu phẩy động định dạng độ chính xác đơn sẽ có 32 bit bao gồm: 1 bit dấu, 23 bit cho phần định trị và 8 bit cho phần số mũ.Lấy ví dụ, 33 bit đầu tiên của số π là 11.001001 00001111 11011010 10100010 0 (lưu ý dấu phẩy cơ số nằm ở sau bit 1 thứ hai từ bên trái qua). Để biểu diễn số π này ở định dạng IEEE 754 độ chính xác đơn, ta phải làm tròn chuỗi bit nói trên còn 24 bit (tại sao lại là 24 bit trong khi phần định trị của định dạng chính xác đơn chỉ có 23 bit sẽ được giải thích ngay sau đây). Để làm tròn như vậy, ta kết hợp các giá trị của bit thứ 24 và bit thứ 25 để được giá trị 11.001001 00001111 11011011. Yêu cầu của chuẩn IEEE 754 là phần định trị phải có giá trị nằm trong khoảng từ 1 đến 2 (tức là chuẩn IEEE 754 quy ước dấu chấm cơ số luôn luôn nằm ngay sau bit 1 đầu tiên). Chính vì thế ta phải lấy tỉ lệ kết quả làm tròn thành 1.1001001 00001111 11011011với số mũ $e = 1$. Đến đây, vì bit đầu tiên (và cũng là duy nhất) đứng trước dấu phẩy cơ số của phần định trị luôn luôn là 1 nên ta không cần lưu trữ bit này và viết gọn phần định trị là 1001001 00001111 11011011. Khi đó, số bit của phần định trị chỉ còn 23 bit khớp với số bit được chuẩn IEEE 754 dùng cho phần định trị. Để xác định giá trị của π, ta dùng công thức:

$$\left(1+\sum_{n=1}^{p-1} bit_n \times 2^{-n}\right) \times 2^e = \left(1+1.2^{-1}+0.2^{-2}+...+1.2^{-23}\right) \times 2^1 = 1.5707964 \times 2$$

với n là bit thứ n của phần định trị. Quá trình lấy tỉ lệ phần định trị sao cho giá trị của nó phải nằm trong khoảng từ 1 đến 2 và bỏ không lưu trữ bit 1 đầu tiên được gọi là chuẩn hóa. Ta có thể xem việc chuẩn hóa giống như một dạng của nén; nó cho phép ta thực hiện lưu trữ 24 bit định trị trong một trường chỉ có 23 bit với lưu ý rằng luôn luôn có một bit 1 ở trước dấu phẩy cơ số.

2.2 Tìm hiểu về chuẩn IEEE 754 – chuẩn dấu phẩy động trong máy tính ngày nay

2.2.1 Chuẩn biểu diễn số thực IEEE 754

Hiệp hội IEEE đã chuẩn hóa cho việc biểu diễn số dấu phẩy động nhị phân trong máy tính bằng cách đưa ra chuẩn IEEE 754. Ngày nay hầu hết các máy tính đều tuân thủ theo chuẩn này. Một số trường hợp ngoại lệ như máy tính lớn IBM và máy vector Cray. Loại máy tính lớn IBM ngoài định dạng thập phân và nhị phân IEEE 754 còn có một định dạng riêng của IBM. Còn với máy vector Cray thì họ T90 có một phiên bản IEEE nhưng máy SV1 vẫn còn dùng định dạng dấu phẩy động của chính Cray.

Chuẩn IEEE 754 đưa ra nhiều định dạng rất gần nhau, chỉ khác nhau ở một ít chi tiết. Năm trong số những định dạng này được gọi là định dạng cơ bản, và hai trong chúng đặc biệt được dùng rộng rãi trong cả phần cứng máy tính và ngôn ngữ lập trình:

Độ chính xác đơn, được gọi bằng tên là "float" trong họ ngôn ngữ lập trình C và tên là "real" hay "real*4" trong ngôn ngữ Fortran. Đây là định dạng nhị phân chiếm 32 bit (4 byte) và phần định trị của nó có độ chính xác 24 bit (tương đương với khoảng 7 chữ số thập phân).

Độ chính xác kép, được gọi bằng tên là "double" trong họ ngôn ngữ lập trình C và tên là "double precision" hay "real*8" trong ngôn ngữ Fortran. Đây là định dạng nhị phân chiếm 64 bit (8 byte) và phần định trị của nó có độ chính xác 53 bit (tương đương với khoảng 16 chữ số thập phân).

Các định dạng khác là nhị phân với độ chính xác bậc bốn (128 bit), cũng như là dấu phẩy động thập phân (64 bit) và dấu phẩy động thập phân "kép" (128 bit).

Các định dạng ít thông dụng hơn:

+ Định dạng độ chính xác mở rộng, mỗi số chiếm 80 bit.

+ Định dạng bán chính xác cũng gọi là dấu phẩy động 16, mỗi số chiếm 16 bit.

Bất kỳ một số nguyên nào có giá trị tuyệt đối nhỏ hơn hay bằng 224 đều có thể biểu diễn một cách chính xác bằng định dạng độ chính xác đơn, và bất kỳ số nguyên nào có giá trị tuyệt đối nhỏ hơn hay bằng 253 cũng có thể biểu diễn một cách chính xác bằng định dạng độ chính xác kép.

Mặc dù các định dạng 32 bit ("đơn") và 64 bit ("kép") hiện nay là phổ biến, nhưng chuẩn IEEE 754 cũng cho phép nhiều mức chính xác khác nhau. Lấy ví dụ, các phần cứng máy tính như họ Pentium Intel và họ 68000 Motorola thường có thêm định dạng độ chính xác mở rộng 80 bit, với phần mũ 15 bit, phần định trị 64 bit (không có bit ẩn) và 1 bit dấu. Một dự án nhằm mục đích sửa đổi chuẩn IEEE 754 đã được khởi động trong năm 2000 (xem IEEE 754 sửa đổi. Dự án này đã hoàn thành và được công nhận vào tháng 6 năm 2008. Nó bao gồm các định dạng dấu phẩy động thập phân và định dạng dấu phẩy động 16 bit ("nữa"). Định dạng 16 bit nhị phân có cùng cấu trúc và quy luật như các định dạng cũ khác với 1 bit dấu, phần mũ 5 bit và 10 bit phần định trị. Định dạng này hiện đang được sử dụng trong ngôn ngữ đồ họa Cg của NVIDIA, và có mặt trong chuẩn mở EXR.

Thông thường thì các số dấu phẩy động được thể hiện trong bộ nhớ máy tính theo thứ tự từ trái sang phải gồm bit dấu, phần mũ, rồi đến phần định trị. Với định dạng nhị phân IEEE 754 chúng thường được biểu diễn bằng các phần sau:

Kiểu	Dấu	Phần mũ	Phần định trị	Tổng số bit	Phân cực mũ	Độ chính xác
Nửa	1	5	10	16	15	11
Đơn	1	8	23	32	127	24
Kép	1	11	52	64	1023	53
Bậc bốn	1	15	112	128	16383	113

Bảng 2.2.1 Các dạng biểu diễn cho chuẩn IEEE 754

Cần lưu ý rằng phần mũ có giá trị âm hay dương, nhưng khi lưu trữ trong máy tính người ta không dùng hệ bù 2 để biểu diễn phần mũ mà lại sử dụng một phương pháp khác: biểu diễn phần mũ dưới dạng một số không dấu nhưng có một giá trị "phân cực" cố định thêm vào. Lấy ví dụ, với định dạng độ chính xác đơn thì giá trị phân cực bằng +127, có nghĩa là để biểu diễn giá trị phần mũ bằng 0 thì người ta lưu trong máy tính các bit là 01111111 tức là +127. Tương tự, ở độ chính xác đơn, giá trị phần mũ là 11111110 (+254) thì giá trị thực phải hiểu là +127 (= 254 – 127). Mỗi định dạng bán chính xác, độ chính xác đơn, độ chính xác kép… đều có một giá trị phân cực riêng của nó…(xem bảng trên). Có hai trường hợp đặc biệt: nếu phần mũ có tất cả các bit bằng 0 thì nó được dùng để biểu diễn các số zero và số không chuẩn hóa, nếu phần mũ có tất cả các bit bằng 1 thì nó được dùng để biểu diễn các vô cực và NaN. Như vậy, giá trị phần mũ của các số chuẩn hóa có tầm nằm trong [-14, 15] ở độ chính xác nữa, [-126, 127] ở độ chính xác đơn, [-1022, 1023] ở độ chính xác kép, hay [-16382, 16383] với độ chính xác bậc bốn. Khi ta nói đến số chuẩn hóa là ta đã loại trừ không xét các số zero, vô cực, NaN và các số không chuẩn hóa.

Cũng cần nhớ rằng, phần định trị trong định dạng nhị phân của IEEE luôn có một bit 1 đầu tiên không được lưu trữ trong máy tính. Nó được gọi là bit "ẩn" hay bit "hiểu ngầm". Chính vì thế, tuy trong máy tính, định dạng độ chính xác đơn có phần định trị gồm 23 bit nhưng ta phải hiểu nó có độ chính xác lên đến 24 bit. Tương tự như vậy, định dạng độ chính xác kép có độ chính xác 53 bit và độ chính xác bậc bốn là 113. Lấy ví dụ, như ta đã chỉ ra ở phần trên, số π, làm tròn đến độ chính xác 24 bit, có: dấu = 0; phần mũ e = 1; phần định trị s = 11001001000011111011011 (bao gồm cả bit ẩn)

Tổng của giá trị phân cực cho phần mũ (127) và giá trị phần mũ (1) là 128 nên chuỗi 8 bit cần lưu trong máy tính của phần mũ phải là: 10000000 Cuối cùng, toàn bộ giá trị của số π được lưu ở định dạng độ chính xác đơn là 0 10000000 10010010000111111011011 (đã bỏ bit ẩn) = 40490FDB (cơ số 16).

2.2.2 Chuẩn IEEE 32 bit sử dụng trong đề tài

Số thực dấu phẩy động được dùng để biểu diễn các số thực trong tính toán khoa học. Tổng quát một số thực X được biểu diễn theo kiểu số dấu phẩy động như sau: $X = M \times R^E$

Trong đó:

- M là phần định trị (Mantissa)
- R là cơ số (Radix)
- E là phần mũ (Exponent)

Đối với chuẩn IEEE 754 - 32 bit, ta có cơ số $R = 2$ và bao gồm 32 bit cụ thể:

Bit dấu	**Phần số mũ**	**Phần định trị**
S	**E**	**M**
1 bit	8 bit	23 bit

Bảng 2.2.2 Chuẩn biểu diễn IEEE 754 – 32 bit

- S là bít dấu (số dương S = 0)
- e là mã excess của phần mũ E (e = E+127 hay E = e-127, số 127 ở đây là độ lệch bias)
- m là phần lẽ của phần định trị M (M = 1.m)

Ví dụ: Chuyển số thực theo chuẩn IEEE 754 – 32 bit sau về lại số thập phân tương ứng:

1	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Ta có:

+ $S = 1$ → số âm

+ $e = 10000010_2 = 130_{10} \Rightarrow E = 130 - 137 = 3$

+ $m = 101011 \Rightarrow M = 1{,}101011$

$\Rightarrow X = -1.101011 \times 2^3 = -1101{,}011 = -13{,}375$

CHƯƠNG III: CÁC MODULE PHỤC VỤ VÀ SƠ ĐỒ KHỐI CỦA BỘ FPU

3.1 Sơ đồ khối tổng quát của hệ thống

Bộ FPU được thiết kế trong đề tài bao gồm 3 ngõ vào lần lượt là:

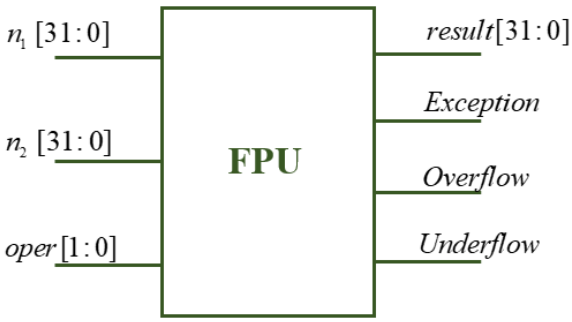
+ n_1, n_2 là hai số đầu vào cần thực hiện tính toán.

+ *oper* là ngõ lựa chọn phép tính cho hai số trên.

Các ngõ ra lần lượt là:

+ *result* là kết quả cuối cùng của phép tính.

+ *Exception, Overflow, Underflow* là các ngõ báo tràn và các trường hợp ngoại lệ.



Hình 3.1 Sơ đồ khối tổng quát của bộ FPU

3.2 Các module phục vụ cho hệ thống

3.2.1 Multiplexer

+ Chức năng: Ứng với mỗi tổ hợp các bit chọn kênh, kênh tương ứng với tổ hợp bit đó sẽ được chọn sang ngõ ra.

+ Nguyên lý: Mỗi bộ tách kênh có hai ngõ vào in0 và in1, bit chọn kênh là sl(select) và ngõ ra là out. Ta biểu diễn thông qua phương trình logic sau: $out = \overline{sl}.in0 + sl.in1$ với sơ đồ khối như hình bên:

Hình 3.2.1 Sơ đồ khối tổng quát của bộ Multiplexer

Từ chương trình trên, ta có thể mở rộng lên thành Mux_8bit, Mux_24bit, Mux_32bit bằng cách kết hợp các module đơn vừa tạo ra trước đó để tăng số bit của các kênh vào.

3.2.2 Reduction_and

+ Chức năng: Phát hiện ra các số thuộc nhóm số ngoại lệ hoặc không phải là số (*Exception or NaN)* đối với trường hợp toàn các bit 1.

+ Nguyên lý: Module được thực hiện bằng cách and tất cả các bit có trong chuỗi nhị phân lại với nhau, do đó ngõ ra module này chỉ bằng 1 khi và chỉ khi tất cả các ngõ vào đều bằng 1.

Hình 3.2.2 Sơ đồ khối tổng quát của bộ Reduction_and

3.2.3 Reduction_or

+ Chức năng: Phát hiện ra các số thuộc nhóm số ngoại lệ hoặc không phải là số (*Exception or NaN)* đối với trường hợp toàn các bit 0.

+ Nguyên lý: Module được thực hiện bằng cách or tất cả các bit có trong chuỗi nhị phân lại với nhau, do đó ngõ ra module này chỉ bằng 0 khi và chỉ khi tất cả các ngõ vào đều bằng 0.

Hình 3.2.3 Sơ đồ khối tổng quát của bộ Reduction_or

Từ module trên, ta mở rộng lên thành các module cho các đầu vào nhiều bit hơn bằng cách kết hợp các module trước đó.

3.2.4 Reduction_nor

+ Chức năng: Kiểm tra xem ở bộ chia, số chia nó bằng 0 hay không.

+ Nguyên lý: Module được thực hiện bằng cách nor tất cả các bit có trong chuỗi nhị phân lại với nhau, do đó ngõ ra module này chỉ bằng 1 khi và chỉ khi tất cả các ngõ vào đều bằng 0. Đoạn chương trình sau minh họa cho module đó:

Hình 3.2.4 Sơ đồ khối tổng quát của bộ Reduction_nor

3.2.5 Complement

+ Chức năng: Thực hiện phép bù 1 một số nhị phân để phục vụ cho phép toán trừ.

+ Nguyên lý: Module này được thực hiện bằng cách đảo tất cả các bit có trong chuỗi nhị phân bằng cổng NOT.

Hình 3.2.5 Sơ đồ khối tổng quát của bộ Complement

3.2.6 Adder

+ Chức năng: Thực hiện cộng hai số nhị phân lại với nhau cùng với số nhớ trước đó để tạo thành kết quả ở ngõ ra cũng như là bit tràn.

+ Nguyên lý: Module này thực hiện việc xử lý từng bit tương ứng từng cột lại với nhau bao gồm cộng các bit cùng với số tràn ở cột trước đó để thu được kết quả tổng ở từng cột và bit tràn.

Hình 3.2.6 Sơ đồ khối tổng quát của bộ Adder

3.2.7 Complement_2s

+ Chức năng: Thực hiện phép bù 2 để phục vụ cho phép toán trừ.

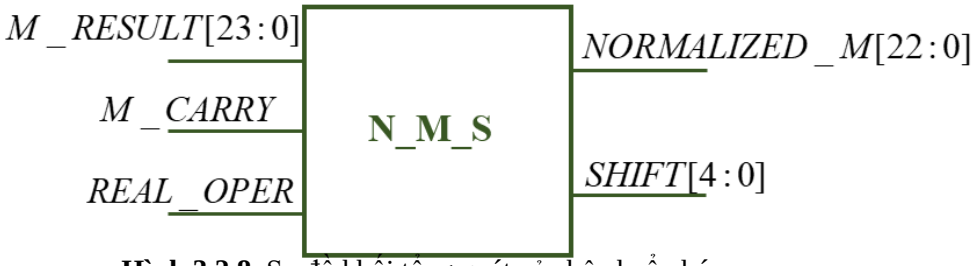
+ Nguyên lý: Phép bù 2 có được bằng cách cộng kết quả của bộ bù 1 cho 1, việc ấy được thực hiện thông qua một bộ cộng.

Hình 3.2.7 Sơ đồ khối tổng quát của bộ Complement_2s

3.2.8 NormalizeMandfindShift

+ Chức năng: Thực hiện việc chuẩn hóa phần định trị theo tiêu chuẩn nhất định.

+ Nguyên lý: Phần định trị đầu vào là một chuỗi nhị phân 24 bit, ta cần biểu diễn chúng về dạng 1X, do đó những chuỗi nhị phân nào không bắt đầu từ 1 ta phải tiến hành dịch trái chuỗi nhị phân đó để thu được dạng chuẩn hóa, đồng thời lưu lại số lượng bit đã dịch để giảm đi phần số mũ tương ứng lượng bit đã dịch đó.



Hình 3.2.8 Sơ đồ khối tổng quát của bộ chuẩn hóa

3.2.9 Multiplier24bit

+ Chức năng: Thực hiện việc nhân hai số nhị phân 24 bit lại với nhau.

+ Nguyên lý: Bộ nhân được hỗ trợ sẵn bởi ngôn ngữ Verilog nên ta không cần phải xử lý các thành phần bên trong của nó.

Hình 3.2.9 Sơ đồ khối tổng quát của bộ Multiplier

3.2.10 Divider24bit

+ Chức năng: Thực hiện việc chia hai số nhị phân lại với nhau.

+ Nguyên lý: Bộ chia được hỗ trợ sẵn bởi ngôn ngữ Verilog nên ta không cần phải xử lý các thành phần bên trong của nó.

Hình 3.2.10 Sơ đồ khối tổng quát của bộ Divider

CHƯƠNG IV: THIẾT KẾ VÀ GIẢI THUẬT CHO CÁC KHỐI TRONG BỘ FPU

4.1 Thiết kế bộ cộng – trừ

4.1.1 Cơ sở phép toán cộng

Ta mô tả phép tính cộng hai số thực dấu phẩy động dưới dạng sau:

$$M_1 * 2^{E_1} + M_2 * 2^{E_2} = M * 2^E$$

Giả sử, cả M_1, M_2 đều đã được chuẩn hóa và nếu $E_1 = E_2$ thì ta chỉ cần cộng hai phần M_1, M_2 lại với nhau và sau đó chuẩn hóa kết quả. Còn nếu $E_1 \neq E_2$ thì ta cần phải dịch chuyển thay đổi phần số mũ của số có số mũ nhỏ hơn và dịch chuyển phần định trị của số có số mũ nhỏ hơn về bên trái. Ví dụ sau minh họa điều đó:

Ta có: $M_1 * 2^{E_1} = 0.111 \times 2^5$ và $M_2 * 2^{E_2} = 0.101 \times 2^3$

Vì $E_1 \neq E_2$ nên ta dịch chuyển M_2 sang phải 2 đơn vị và tăng thêm 2 đơn vị vào số mũ, khi đó:

$$M_2 * 2^{E_2} = 0.101 \times 2^3 = 0.00101 \times 2^5$$

Lúc này, hai số mũ đã bằng nhau nên ta tiến hành cộng phần định trị lại với nhau:

$$0.111 \times 2^5 + 0.00101 \times 2^5 = (0.111 + 0.00101) \times 2^5 = 01.00001 \times 2^5$$

Tuy nhiên, lúc này phần định trị đã xảy ra hiện tượng tràn *overflow* nên ta chuẩn hóa bằng cách dịch phải phần định trị 1 đơn vị và tăng 1 đơn vị cho phần số mũ, kết quả cuối cùng lúc này là:

$$M \times 2^E = 0.100001 \times 2^6$$

Tóm tắt lại, đối với phép cộng ta có:

+ **Bước 1:** Xác định số có số mũ lớn hơn trong hai số này để làm căn cứ cho quá trình tính toán.

+ **Bước 1:** Chuyển hai số thực về cùng một mũ (exponent) bằng cách dịch chuyển chấm động đến khi cả hai số có cùng mũ.

+ **Bước 2:** Thực hiện phép cộng trên phần định trị của hai số thực.

+ **Bước 3:** Xử lý kết quả:

- Kiểm tra và xử lý tràn số nếu có.
- Khi tổng hai định trị bị tràn thì dịch phải nó sang 1 đơn vị và tăng 1 đơn vị cho số mũ.
- Khi kết quả phần định trị chưa chuẩn hóa thì dịch trái nó sang X đơn vị và giảm đi X đơn vị cho phần số mũ.

4.1.2 Cơ sở phép toán trừ

Ta mô tả phép tính trừ hai số thực dấu phẩy động dưới dạng sau:

$$M_1 * 2^{E_1} - M_2 * 2^{E_2} = M * 2^E$$

Tuy nhiên, để thực hiện được phép tính trừ, ta cần phải chuyển đổi nó về phép toán cộng với số bù 2 của số trừ, mà số bù 2 nó được bằng cách cộng thêm 1 vào số bù 1 của số nhị phân đó. Ta minh họa bởi phương trình sau:

$$A - B = A + (B)_2 = A + (B)_1 + 1$$

Trong đó:

+ Phép bù 1: Là một phép toán trong hệ thống số nhị phân, được sử dụng trong các thuật toán tính toán trên số dấu phẩy động, trong đó ta cần thực hiện phép trừ bằng phương pháp bù 2. Khi thực hiện phép bù 1 trên một số nhị phân, ta đảo ngược tất cả các bit trong số đó. Nói cách khác, mỗi số 0 trong số ban đầu được thay bằng 1, và mỗi số 1 trong số ban đầu được thay bằng 0. Ví dụ, nếu ta muốn tính bù 1 của số nhị phân 10101, ta sẽ đảo ngược nó thành 01010.

+ Phép bù 2: Được thực hiện từ việc cộng thêm 1 đơn vị vào kết quả của phép bù 1. Chẳng hạn, ta có bù 2 của số nhị phân 10101 sẽ được tính bằng $01010 + 1 = 01011.$

Như vậy, từ phép toán trừ ban đầu ta đã chuyển sang phép tính cộng đã được đề cập trước đó. Các bước làm tương tự phép tính cộng, tuy nhiên kết quả cuối cùng của phép cộng bù 2 này lại chia thành hai trường hợp sau:

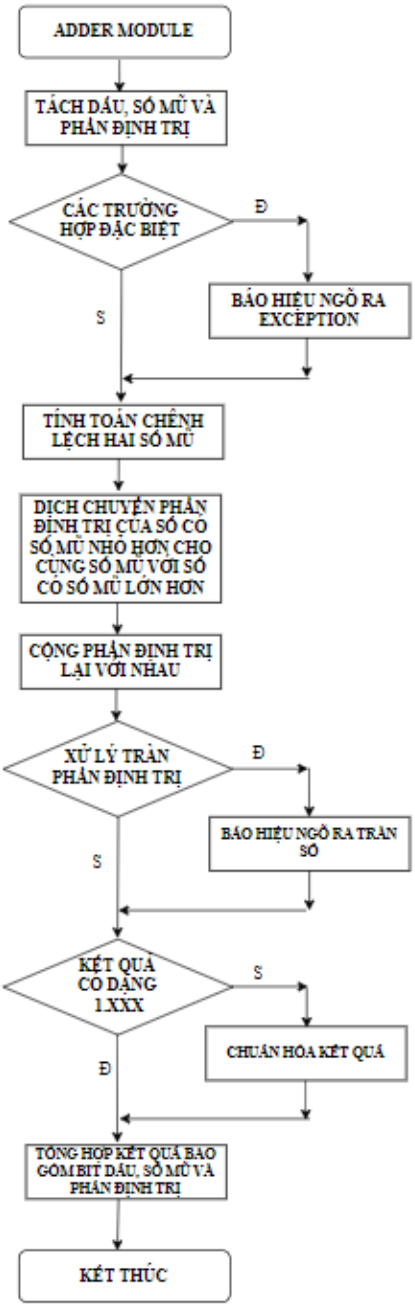
+ Nếu sau khi cộng với bù 2 mà có bit tràn được sinh ra, thì chắc chắn kết quả của phép trừ là số dương, và kết quả thu được từ phép cộng bù 2 chính là kết quả cuối cùng của phép trừ.

+ Nếu sau khi cộng với bù 2 mà không phát sinh bit tràn, thì chắc chắn kết quả của phép trừ là số âm, và kết qua thu được từ phép cộng bù 2 phải được lấy bù 2 thêm một lần nữa để tạo thành kết quả cuối cùng.

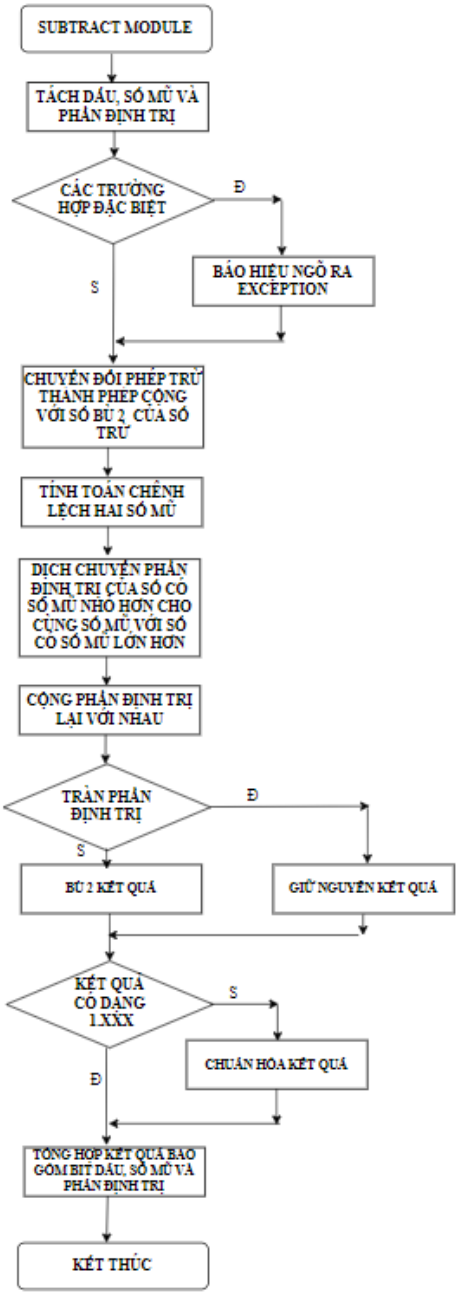
Tóm lại, tùy thuộc vào bit tràn được sinh ra từ bộ cộng bù 2 mà ta lựa chọn kết quả cuối cùng là kết quả trước đó hay kết quả được bù 2 của kết quả trước đó.

4.1.3 Giải thuật cho phép tính cộng – trừ

Ta minh họa quá trình thực hiện các phép toán cộng trừ bởi lưu đồ sau:



Hình 4.1.3.1 Lưu đồ cho phép toán cộng



Hình 4.1.3.2 Lưu đồ cho phép toán trừ

4.2 Thiết kế bộ nhân

4.2.1 Cơ sở phép toán nhân

Ta mô tả phép tính nhân hai số thực dấu phẩy động dưới dạng sau:

$$M_1 * 2^{E_1} \times M_2 * 2^{E_2} = (M_1 M_2) * 2^{E_1+E_2}$$

Như vậy, đối với phần số mũ ta sẽ cộng chúng lại với nhau và phần định trị sẽ được nhân lại, ví dụ:

$$0.100 * 2^3 \times 0.100 * 2^4 = 0.010 * 2^{3+4}$$

Tuy nhiên, đối với kết quả trên thì phần định trị chưa được chuẩn hóa nên ta chuẩn hóa lại kết quả thành: $0.100 * 2^{3+4-1}$. Khi đó, ta đã thực hiện chuẩn hóa bằng cách dịch trái phần định trị một

đơn vị và trừ đi một đơn vị vào phần số mũ. Kết thúc, nếu phần số mũ không nằm trong dải cho phép

của phần số mũ trong hệ thống thì sẽ xảy ra trường hợp tràn bít phần số mũ, 8 bit số mũ trong tiêu chuẩn IEEE 754-32 bit cho phép biểu diễn các giá trị trong dải từ -128 đến 127. Trong đó, bit đầu tiên được sử dụng để biểu diễn dấu, vì vậy chỉ còn lại 7 bit để biểu diễn phần số mũ. Do đó, số mũ được biểu diễn dưới dạng số nguyên có dấu 2's complement, với số âm được biểu diễn bằng cách lấy bù hai của số dương tương ứng, khi xảy ra tràn bít thì sẽ có tín hiệu báo tràn.

Tóm lại, khi nhân hai số thực dấu phẩy động, ta thực hiện:

+ **Bước 1:** Xác định dấu của kết quả nhân bằng cách XOR dấu của a và b.

+ **Bước 2:** Cộng chỉ số mũ của a và b lại với nhau. Trừ đi giá trị trung bình (127) để tính ra giá trị của chỉ số mũ. Kiểm tra xem phần mũ của số kết quả có vượt quá giới hạn cho phép hay không (giá trị lớn nhất là 255). Nếu vượt quá, ta sẽ trả về giá trị "vô cực" hoặc "báo lỗi", tùy vào ý định sử dụng của người dùng.

+ **Bước 3:** Lấy phần định trị của hai số và thêm bit 1 vào vị trí đầu tiên của dãy nhị phân, sau đó nhân chúng lại với nhau. Kết quả này sẽ cho ta phần định trị của số kết quả.

+ **Bước 4:** Kiểm tra xem phần trị của số kết quả có dạng "1.xxx..." không (trong đó "x" là các số 0 hoặc 1). Nếu có, ta cần dịch trái phần trị và tăng phần mũ đi một đơn vị để đưa phần trị về dạng "0.xxx...".

+ **Bước 5:** Cuối cùng, ta sẽ xây dựng lại số kết quả dựa trên phần dấu, phần mũ và phần trị đã tính được ở các bước trên.

Ví dụ: Giả sử ta cần tính tích của hai số thực dấu phẩy động chuẩn IEEE 754 - 32 bit sau đây:

$a = -1.25 = 1\ 01111111\ 01000000000000000000000$

$b = 0.5 = 0\ 01111110\ 00000000000000000000000$

Theo các bước trên, ta có thể tính được kết quả như sau:

+ Phần dấu của số kết quả là "1" (tương ứng với phép XOR giữa phần dấu của a và b).

+ Phần mũ của số kết quả là $(124 + 120) - 127 = 117$ (tương ứng với tổng phần mũ của a và b trừ đi giá trị trừ).

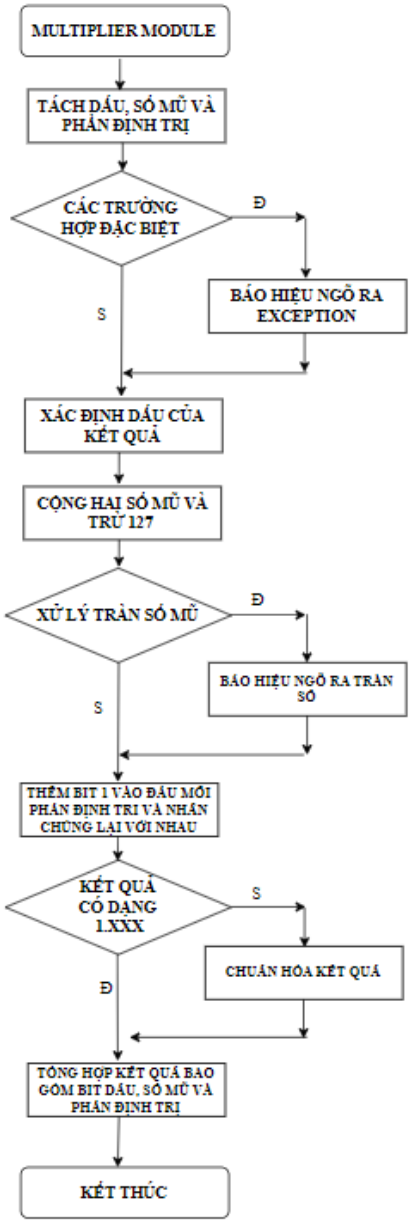
+ Phần định trị của số kết quả là $(1.01... * 2^{0}) * (1.00... * 2^{-1}) = 1.010... * 2^{-1}$.

+ Vì phần trị của số kết quả có dạng "1.xxx...", nên ta cần dịch trái phần trị và tăng phần mũ lên một đơn vị để đưa phần trị về dạng "0.xxx...". Do đó, phần trị của số kết quả sẽ là 0.101... và phần mũ sẽ là 126.

+ Số kết quả sẽ là: $1\ 01111110\ 01010000000000000000000$ (tương ứng với phần dấu là 1, phần mũ là 126 và phần trị là 0.101...

+ Vậy, tích của hai số a và b sẽ là -0.625, được biểu diễn bởi số thực dấu phẩy động chuẩn IEEE 754 - 32 bit là $1\ 01111110\ 01010000000000000000000$.

4.2.2 Giải thuật cho phép tính nhân



Hình 4.2.2 Lưu đồ cho phép toán nhân

4.3 Thiết kế bộ chia

4.3.1 Cơ sở phép toán chia

Ta mô tả phép tính chia hai số thực dấu phẩy động dưới dạng sau:

$$\left(M_1 * 2^{E_1}\right) : \left(M_2 * 2^{E_2}\right) = \left(M_1 : M_2\right) * 2^{E_1 - E_2}$$

Đối với phần số mũ ta sẽ trừ chúng lại với nhau và phần định trị sẽ được chia nhau, ví dụ:

$$\left(0.010 * 2^{7}\right) : \left(0.100 * 2^{4}\right) = \left(0.010 : 0.100\right) * 2^{7-4} = 0.100 * 2^{3}$$

Kết thúc nếu phần số mũ không nằm trong dải cho phép của phần số mũ trong hệ thống thì sẽ xảy ra trường hợp tràn bít phần số mũ. Đối với chuẩn số thực IEEE 754 – 32 bit, ta sử dụng 8 bit cho phần số mũ thì dãy giá trị cho phép của phần mũ là từ 1000 0000 cho đến 0111 1111 (từ -128 đến 127) , khi xảy ra tràn bít thì sẽ có tín hiệu báo tràn.

Tóm lại, khi chia hai số thực dấu phẩy động, ta thực hiện:

+ **Bước 1:** Xác định dấu của kết quả chia bằng cách XOR dấu của a và b.

+ **Bước 2:** Lấy phần mũ của số bị chia và phép mũ của số chia, trừ chúng với nhau và cộng lại giá trị "127" (giá trị dùng để cộng trong bảng IEEE 754 - 32 bit).

+ **Bước 3:** Kết quả của phép tính này sẽ cho ta phần mũ của số kết quả. Kiểm tra xem phần mũ của số kết quả có nhỏ hơn giới hạn cho phép hay không (giá trị nhỏ nhất là 0). Nếu nhỏ hơn, ta sẽ trả về giá trị "0" hoặc "báo lỗi", tùy vào ý định sử dụng của người dùng.

+ **Bước 4:** Lấy phần định trị của số bị chia và thêm một vào vị trí đầu tiên của dãy nhị phân, sau đó chia chúng cho phần trị của số chia. Kết quả này sẽ cho ta phần trị của số kết quả.

+ **Bước 5:** Kiểm tra xem phần định trị của số kết quả có dạng "1.xxx..." không (trong đó "x" là các số 0 hoặc 1). Nếu có, ta cần dịch trái phần trị và tăng phần mũ đi một đơn vị để đưa phần trị về dạng "0.xxx...".

+ **Bước 6:** Cuối cùng, ta sẽ xây dựng lại số kết quả dựa trên phần dấu, phần mũ và phần trị đã tính được ở các bước trên.

Ví dụ: Giả sử ta cần tính tích của hai số thực dấu phẩy động chuẩn IEEE 754 - 32 bit sau đây:

$a = -1.25 = 1\ 01111111\ 01000000000000000000000$

$b = 0.5 = 0\ 01111110\ 00000000000000000000000$

+ Phần dấu của số kết quả là "1" (tương ứng với phép XOR giữa phần dấu của a và b).

+ Phần mũ của số kết quả là $127 + (127 - 126) = 128$ (tương ứng với hiệu phép mũ của a và b cộng với giá trị cộng).

+ Phần trị của số kết quả là $(1.01... * 2^{0}) / (1.00... * 2^{-1}) = 1.010... * 2^{-1}$.

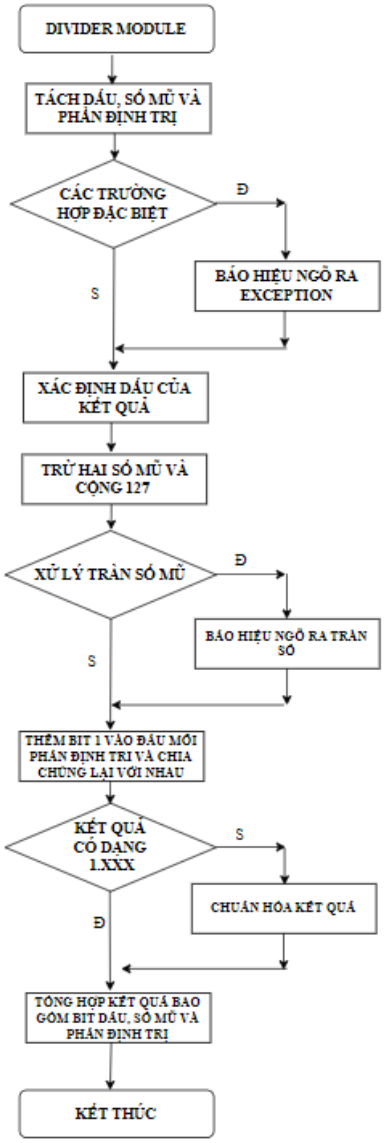
+ Vì phần trị của số kết quả có dạng "1.xxx...", nên ta cần dịch trái phần trị và tăng phần mũ lên một đơn vị để đưa phần trị về dạng "0.xxx...". Do đó, phần trị của số kết quả sẽ là 0.101... và phần mũ sẽ là 128.

+ Kiểm tra xem phần mũ của số kết quả có nhỏ hơn giới hạn cho phép hay không. Trong trường hợp này, phần mũ của số kết quả không nhỏ hơn giới hạn cho phép, nên ta có thể tiếp tục tính toán.

+ Số kết quả sẽ là: $1\ 10000000\ 01000000000000000000000$ (tương ứng với phần dấu là 1, phần mũ là 132 và phần trị là 0.101...).

+ Vậy, thương của hai số a và b sẽ là -2.5, được biểu diễn bởi số thực dấu phẩy động chuẩn IEEE 754 - 32 bit là $1\,10000000\,01000000000000000000000$.

4.3.2 Giải thuật cho phép tính chia

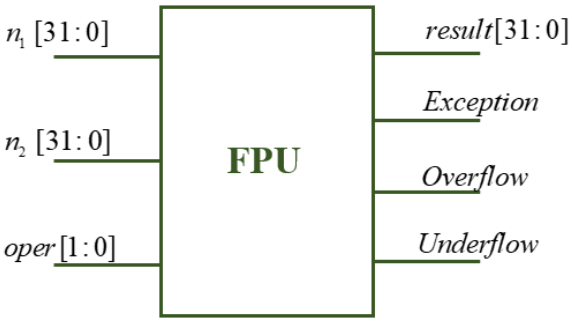


Hình 4.3.2 Lưu đồ cho phép toán chia

4.4 Tổng hợp các module

4.4.1 Tổng hợp các thiết kế

Sau khi thiết kế và xây dựng các module phục vụ cũng như các module tính toán chính trong chương trình ta tiến hành tổng hợp chúng lại với nhau để tạo thành một module hoàn chỉnh với sơ đồ khối như hình bên:



Hình 4.4.1.1 Sơ đồ khối tổng quát của bộ FPU

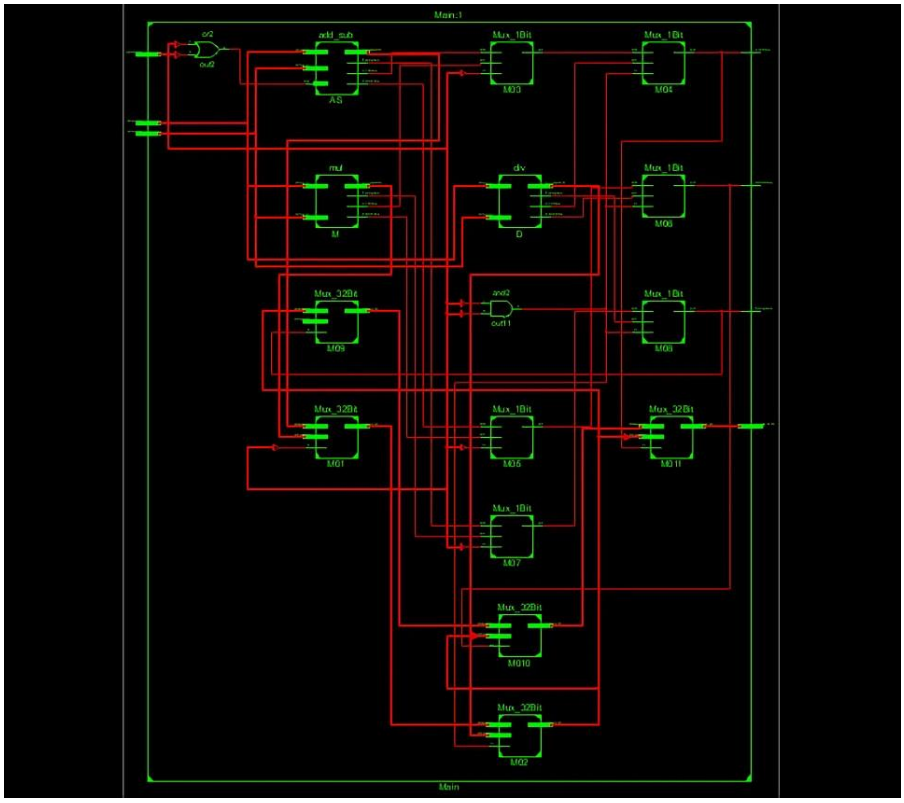
Module sau cùng của chúng ta bao gồm 3 ngõ vào và 4 ngõ ra. Các ngõ vào bao gồm hai số thực dấu phẩy động cần thực hiện tính toán, một ngõ vào chọn phép tính, đối với 4 phép tính ta sử dụng 2 bit nhị phân để phân biệt. Các ngõ ra bao gồm kết quả của phép tính và các ngõ báo tràn số cũng như số ngoại lệ.

Tiến hành nối dây các module tính toán lại với nhau:

Sử dụng các bộ ghép kênh để phân biệt ngõ ra kết quả cũng như các cờ báo tràn, báo ngoại lệ giữa các phép tính kết hợp với ngõ vào chọn *oper*.

Cuối cùng là xử lý kết quả ngõ ra khi gặp phải các trường hợp tràn hay ngoại lệ:

Sau đây là sơ đồ nối dây của hệ thống:



Hình 4.4.1.2 Sơ đồ nối dây của hệ thống

4.4.2 Nguyên lý thực thi các phép tính

Dựa vào chương trình của bộ ghép kênh, ta có thể phân biệt các phép tính thông qua bảng sau:

Oper	Phép toán được thực hiện
00	Phép cộng
01	Phép trừ
10	Phép nhân
11	Phép chia

Bảng 4.4.2 Mã oper cho các phép tính

CHƯƠNG V: ĐÁNH GIÁ KẾT QUẢ ĐẠT ĐƯỢC QUA TEST BENCH

5.1 Mô hình testbench tổng quát

Chương trình sau mô tả cho testbench để mô phỏng hệ thống đã thiết kế:

```
            oper = 2'd0; #50;

            $display("Addtion result : %b",result);
            $display("Overflow : %b , Underflow : %b , Exception : %b",Overflow,Underflow,Exception);

            oper = 2'd1; #50;

            $display("Subtraction result : %b",result);
            $display("Overflow : %b , Underflow : %b , Exception : %b",Overflow,Underflow,Exception);

            oper = 2'd2; #50;

            $display("Multiplication result : %b",result);
            $display("Overflow : %b , Underflow : %b , Exception : %b",Overflow,Underflow,Exception);

            oper = 2'd3; #50;

            $display("Division result : %b",result);
            $display("Overflow : %b , Underflow : %b , Exception : %b",Overflow,Underflow,Exception);



        end

endmodule
```

```
        initial begin
                // Initialize Inputs
                n1 = 32'b01000011000011111000111101011100;    // 143.56
                n2 = 32'b11000010101011101101111110111110;    // -87.437
```

5.2 Mô tả các testcase

+ Khởi tạo ban đầu:

```
            initial begin
                    // Initialize Inputs
                    n1 = 32'b01000011000011111000111101011100;    // 143.56
                    n2 = 32'b11000010101011101101111110111110;    // -87.437
```

Để thực hiện được các phép tính, ta cần phải khởi tạo các số đầu vào dưới dạng chuẩn IEEE 754 – 32 bit, ở trong ví dụ này ta đưa các toán hạng vào lần lượt là $n_1 = 143.56$ và $n_2 = -87.437$ và được chuyển sang dạng chuẩn IEEE 754 – 32 bit ta được hai chuỗi nhị phân 32 bit như trên. Vì module được thiết kế thực hiện các phép toán cộng, trừ, nhân, chia. Như vậy, để chỉ định thực

hiện một trong số bốn phép toán trên, ta cần phải sử dụng 2 bit nhị phân để tạo thành tổ hợp chọn cho mỗi phép toán, cụ thể:

+ Thực hiện phép toán cộng:

```
oper = 2'd0; #50;

$display("Addtion result : %b",result);
$display("Overflow : %b , Underflow : %b , Exception : %b",Overflow,Underflow,Exception);
```

Để thực hiện phép tính cộng, ta sử dụng tổ hợp chọn $oper = 2'd0 = 2'b00$, và sau đó sử dụng lệnh *display* để in giá trị của kết quả tính toán sang màn hình. Cụ thể ở đây ta sẽ in ra giá trị của kết quả phép tính cộng (*result)*, giá trị các cờ tràn *underflow*, *overflow* và ngõ báo số ngoại lệ exception.

+ Thực hiện phép toán trừ:

```
oper = 2'd1; #50;

$display("Subtraction result : %b",result);
$display("Overflow : %b , Underflow : %b , Exception : %b",Overflow,Underflow,Exception);
```

Để thực hiện phép tính trừ, ta sử dụng tổ hợp chọn $oper = 2'd1 = 2'b01$, và sau đó sử dụng lệnh *display* để in giá trị của kết quả tính toán sang màn hình. Cụ thể ở đây ta sẽ in ra giá trị của kết quả phép tính trừ (*result)*, giá trị các cờ tràn *underflow*, *overflow* và ngõ báo số ngoại lệ exception.

+ Thực hiện phép toán nhân:

```
oper = 2'd2; #50;

$display("Multiplication result : %b",result);
$display("Overflow : %b , Underflow : %b , Exception : %b",Overflow,Underflow,Exception);
```

Để thực hiện phép tính nhân, ta sử dụng tổ hợp chọn $oper = 2'd2 = 2'b10$, và sau đó sử dụng lệnh *display* để in giá trị của kết quả tính toán sang màn hình. Cụ thể ở đây ta sẽ in ra giá trị của kết quả phép tính nhân (*result)*, giá trị các cờ tràn *underflow*, *overflow* và ngõ báo số ngoại lệ exception.

+ Thực hiện phép toán chia:

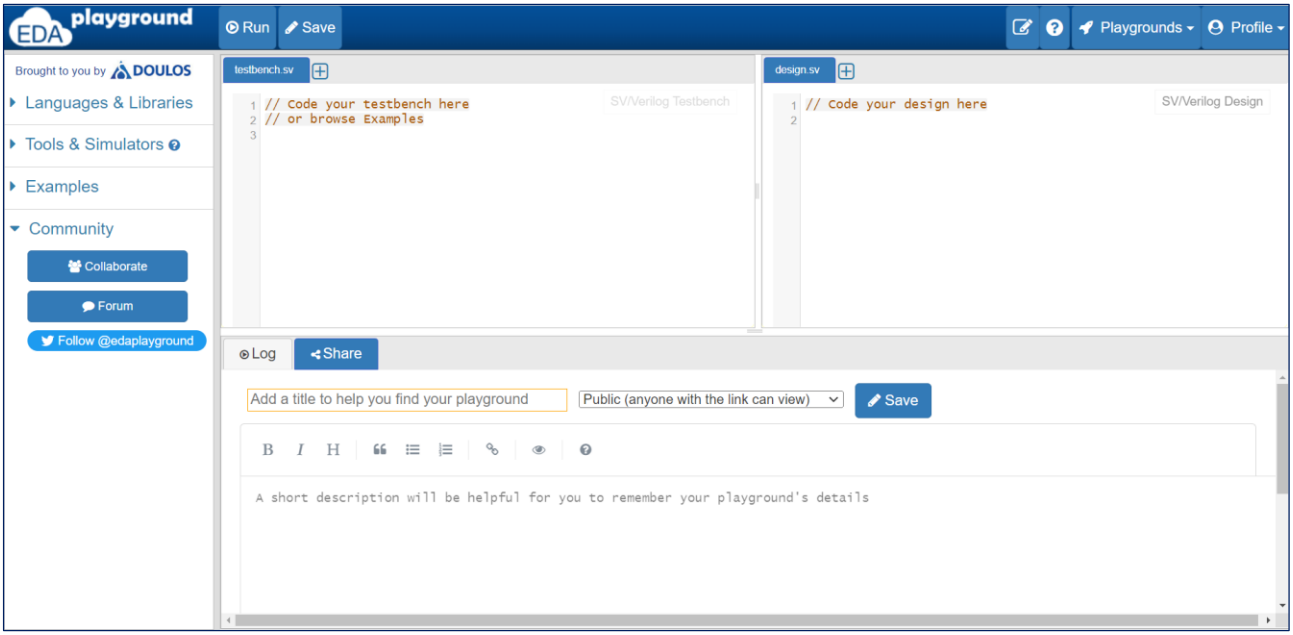
```
oper = 2'd3; #50;

$display("Division result : %b",result);
$display("Overflow : %b , Underflow : %b , Exception : %b",Overflow,Underflow,Exception);
```

Để thực hiện phép tính nhân, ta sử dụng tổ hợp chọn $oper = 2'd3 = 2'b11$, và sau đó sử dụng lệnh *display* để in giá trị của kết quả tính toán sang màn hình. Cụ thể ở đây ta sẽ in ra giá trị của kết quả phép tính chia (*result)*, giá trị các cờ tràn *underflow*, *overflow* và ngõ báo số ngoại lệ exception.

5.3 Kết quả đạt được

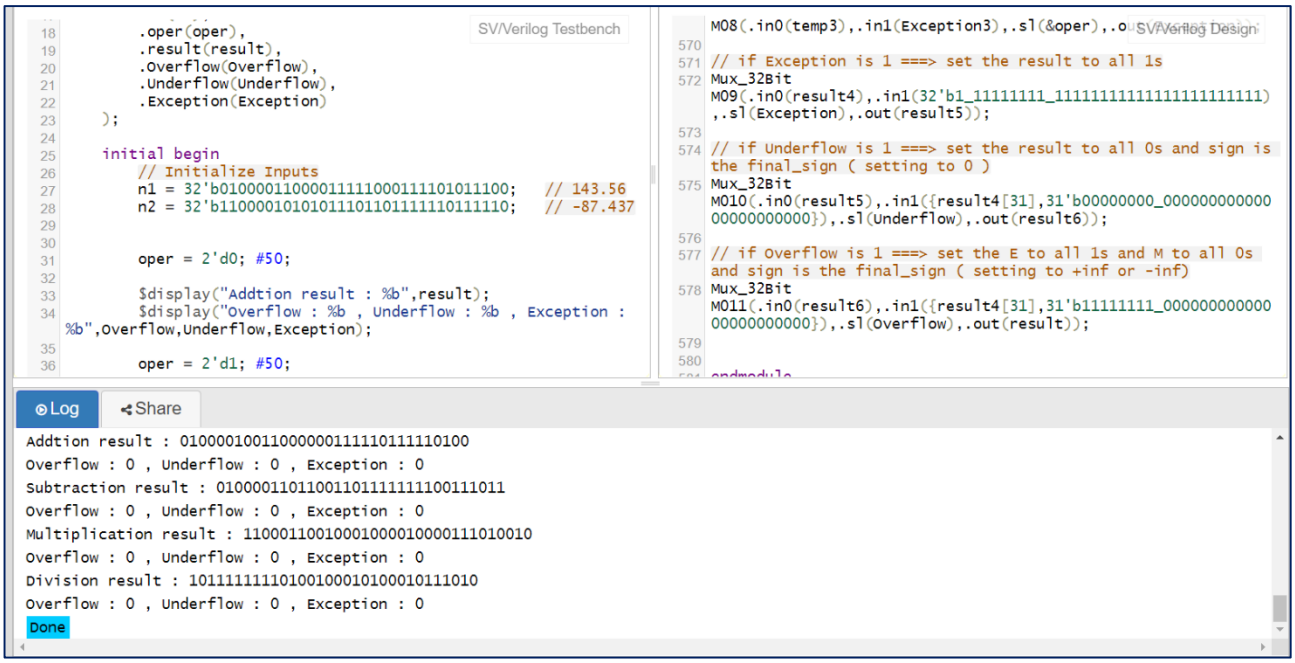
Tiến hành viết chương trình chính cũng như chương trình testbench vào trang web EDA Playground – trang web hỗ trợ trình biên dịch cho ngôn ngữ Verilog và mô phỏng được kết quả của chương trình cần thực thi:



Hình 5.3.1 Cửa sổ chính của trang web EDAplayground

Ở khoảng trắng bên phải *design.sv* ta sẽ nhập chương trình mô tả hệ thống vào, còn ở khoảng trắng bên trái *testbench.sv* chính là nơi ta nhập chương trình testbench vào.

Sau khi nhập chương trình vào, ta tiến hành bấm RUN để xem kết quả mô phỏng, khi đó ta thu được:



Hình 5.3.2 Chương trình và kết quả mô phỏng

Như vậy, kết quả mô phỏng cho ta thấy được kết quả cũng như giá trị của các cờ tràn, cờ báo ngoại lệ của từng phép tính cộng, trừ, nhân, chia.

```
Addtion result : 01000010011000000111110111110100
Overflow : 0 , Underflow : 0 , Exception : 0
Subtraction result : 01000011011001101111111100111011
Overflow : 0 , Underflow : 0 , Exception : 0
Multiplication result : 11000110010001000010000111010010
Overflow : 0 , Underflow : 0 , Exception : 0
Division result : 10111111110100100010100010111010
Overflow : 0 , Underflow : 0 , Exception : 0
```

Hình 5.3.3 Các kết quả tính toán và giá trị các cờ

5.4 Nhận xét

Với hai số đầu vào là:

$$n_1 = 01000011000011111000111101011100 \quad //143.56$$

$$n_2 = 11000010101011101101111110111110 \quad //-87.437$$

Ta tiến hành kiểm tra lại kết quả mô phỏng của từng phép tính. Thêm vào đó, các số đầu vào đều là các số thỏa mãn không phải là số ngoại lệ, hay khi thực hiện các phép toán cộng, trừ, nhân, chia chúng lại với nhau thì kết quả luôn tồn tại trong phạm vi cho phép biểu diễn được của chuẩn IEEE 754 – 32 bit. Do đó, sẽ không xảy ra trường hợp các số đầu vào là ngoại lệ hay kết quả phép

tính rơi vào hiện tượng tràn trên, tràn dưới. Như vậy, các kết quả in ra *Exception*, *Overflow*, *Undeflow* luôn ở **mức 0.**

5.4.1 Đối với phép toán cộng:

+ Kết quả thu được từ chương trình là: 01000010011000000111110111110100, chuyển kết quả trên về lại thập phân ta được:

s	e								m																						
0	1	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	1	1	1	1	1	0	1	1	1	1	1	0	1	0	0

Bảng 5.4.1 Kết quả phép tính cộng

+ $s = 0$ → kết quả phép tính dương

+ $e = 10000100_2 = 132_{10} \Rightarrow E = 132 - 127 = 5$

+ $M = 11000000111110111110100$

$$\Rightarrow X = 1.M \times 2^E = 1.11000000111110111110100 \times 2^5$$
$$= 111000.000111110111110100_2$$
$$= 56{,}1230010986$$

Đối chiếu lại kết quả thực tế: $n_1 + n_2 = 143.56 + (-87.437) = 56.123$

→ Như vậy, kết quả thu được trên thực tế và chương trình rất giống nhau.

5.4.2 Đối với phép toán trừ:

+ Kết quả thu được từ chương trình là: 01000011011001101111111100111011, chuyển kết quả trên về lại thập phân ta được:

s	e								m																						
0	1	0	0	0	0	1	1	0	1	1	0	0	1	1	0	1	1	1	1	1	1	1	1	0	0	1	1	1	0	1	1

Bảng 5.4.2 Kết quả phép tính trừ

+ $s = 0$ → kết quả phép tính dương.

+ $e = 100000110_2 = 134_{10} \Rightarrow E = 134 - 127 = 7$

+ $M = 11001101111111100111011$

$$\Rightarrow X = 1.M \times 2^E = 1.11001101111111100111011 \times 2^7$$
$$= 11100110.11111100111011_2$$
$$= 230{,}9879760742$$

Đối chiếu lại kết quả thực tế: $n_1 - n_2 = 143.56 - (-87.437) = 230.997$

→ Như vậy, kết quả thu được trên thực tế và chương trình khá giống nhau, sở dĩ có sự sai lệch vì số bit biểu diễn chưa đủ lớn để đảm bảo tính trung thực tuyệt đối cho kết quả.

5.4.3 Đối với phép toán nhân:

+ Kết quả thu được từ chương trình là: 11000110010001000010000111010, chuyển kết quả trên về lại thập phân ta được:

s	e								m																						
1	1	0	0	0	1	1	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	1	1	1	0	1	0	0	1	0

Bảng 5.4.3 Kết quả phép tính nhân

+ $s = 1$ → kết quả phép tính là âm.

+ $e = 10001100_2 = 140_{10} \Rightarrow E = 140 - 127 = 13$

+ $M = 10001000010000111010$

$$\Rightarrow X = -1.M \times 2^E = 1.10001000010000111010 \times 2^{13}$$
$$= -11000100001000.0111010_2$$
$$= -12552,453125$$

Đối chiếu lại kết quả thực tế: $n_1 \times n_2 = 143.56 \times (-87.437) = -12552,45572$

→ Như vậy, kết quả thu được trên thực tế và chương trình khá giống nhau, sở dĩ có sự sai lệch vì số bit biểu diễn chưa đủ lớn để đảm bảo tính trung thực tuyệt đối cho kết quả.

5.4.4 Đối với phép toán chia:

+ Kết quả thu được từ chương trình là: 10111111110100100010100010111010, chuyển kết quả trên về lại thập phân ta được:

s	e								m																						
1	0	1	1	1	1	1	1	1	1	0	1	0	0	1	0	0	0	1	0	1	0	0	0	1	0	1	1	1	0	1	0

Bảng 5.4.4 .1 Kết quả phép tính chia

+ $s = 1$ → kết quả phép tính là âm.

+ $e = 01111111_2 = 127_{10} \Rightarrow E = 127 - 127 = 0$

+ $M = 10100100010100010111010$

$$\Rightarrow X = -1.M \times 2^E = -1.10100100010100010111010 \times 2^0$$
$$= -1.10100100010100010111010_2$$
$$= -1.6418678761$$

Đối chiếu lại kết quả thực tế: $n_1 : n_2 = 143.56 : (-87.437) = -1,641867859$

→ Như vậy, kết quả thu được trên thực tế và chương trình khá giống nhau, sở dĩ có sự sai lệch vì số bit biểu diễn chưa đủ lớn để đảm bảo tính trung thực tuyệt đối cho kết quả.

Ta có bảng tóm tắt các kết quả như sau:

OPER	CALCULATION	ACTUAL RESULTS	SIMULATION	DEVIATION
00	Addition	56,123	56,1230010986	$1,96\times10^{-6}$ %
01	Subtraction	230.997	230,9879760742	$3,9\times10^{-3}$ %
10	Multiplication	−12552,45572	−12552,453125	$2,06\times10^{-5}$ %
11	Devision	−1,641867859	−1.6418678761	$1,04\times10^{-6}$ %

Bảng 5.4.4.2 Tổng hợp các kết quả và sai số

5.5 Các trường hợp ngoại lệ

Để tăng thêm tính trực quan cho hệ thống, ta tiến hành khảo sát thêm trường hợp thử nghiệm với đầu vào ngoại lệ nhằm kiểm tra hoạt động của các ngõ ra *Oveflow*, *Undeflow* và *Exception.*

```
initial begin
    // Initialize Inputs
    n1 = 32'b01111111100011111000111101011100;
    n2 = 32'b11000010101011011011111110111110;
```

Hình 5.5.1 Trường hợp ngoại lệ phần mũ toàn bit 1

Ở đây, ta chọn $n_1 = 01111111100011111000111101011100$ với các bit phần số mũ toàn bit 1, quan sát kết quả mô phỏng của chương trình.

```
Addtion result : 11111111111111111111111111111111
Overflow : 0 , Underflow : 0 , Exception : 1
Subtraction result : 11111111111111111111111111111111
Overflow : 0 , Underflow : 0 , Exception : 1
Multiplication result : 11111111100000000000000000000000
Overflow : 1 , Underflow : 0 , Exception : 1
Division result : 11111111111111111111111111111111
Overflow : 0 , Underflow : 0 , Exception : 1
```

Hình 5.5.2 Các kết quả tính toán và giá trị các cờ trong trường hợp ngoại lệ

Như vậy, cờ báo số ngoại lệ *Exception* trong cả bốn trường hợp trên đều lên mức 1, báo hiệu số đầu vào ngoại lệ do tồn tại 8 bit 1 tương ứng phần số mũ của số n_1. Ngoài ra, ở phép nhân còn xảy ra thêm trường hợp tràn trên *Overflow* đối với số mũ. Vì phát hiện được các trạng thái ngoại lên trên do đó ngõ ra *result* cũng đã được chuẩn hóa về đúng định dạng như đã được quy định trong chương trình.

CHƯƠNG VI: KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

6.1 Đánh giá và nhận xét

Thông qua việc đánh giá và kiểm tra dựa trên testbench, ta nhận thấy rằng:

+ Các phép toán được thực hiện luân phiên nhau tùy vào mã lệnh oper ta đưa vào bộ tính toán.

+ Các kết quả được in ra với độ chính xác theo chuẩn IEEE 754 – 32 bit lên tới 24 bit.

+ Các giá trị cờ báo tràn *overflow, underflow* cũng như cờ báo ngoại lệ cũng được in ra màn hình rất chính xác, với mức logic 0 khi bình thường và mức logic 1 khi phép tính rơi vào các trường hợp đặc biệt đó.

+ Cả bốn phép toán đều cho ra kết quả tính toán rất chính xác so với thực tế, với sai số hầu như là gần 0%. Cụ thể, với hai số đại diện như trên ta tính toán được đối với phép cộng sai số chỉ vỏn vẹn $1,96\times10^{-6}$ %, đối với phép trừ là $3,9\times10^{-3}$ %, cũng như phép nhân là $2,06\times10^{-5}$ % và phép chia là $1,04\times10^{-6}$ %. Các con số trên đã cho ta thấy được sự thành công cũng như là độ chính xác mà hệ thống mang lại.

Nhìn chung, đề tài đã đạt được những mục tiêu cơ bản đã đặt ra trước đó cũng như đã xây dựng gần như hoàn chỉnh khối tính toán số thực dấu phẩy động. Song còn nhiều hạn chế trong đề tài như:

+ Chương trình còn khá dài dòng do tồn tại một số module phục vụ chưa được tối ưu hóa, một số giải thuật chưa được tinh giản và chưa đảm bảo tính chặt chẽ, bao quát tất cả các trường hợp có thể xảy ra trong quá trình tính toán.

+ Số lượng phép tính còn giới hạn trong phạm vi 4 phép tính: cộng, trừ, nhân, chia.

+ Độ chính xác chưa quá hoàn hảo do số bit biểu diễn các số thực chỉ giới hạn trong phạm vi 32 bit.

6.2 Hướng phát triển của đề tài

Nhằm khai thác triệt để những ưu điểm nổi bật đồng thời khắc phục và tối ưu những hạn chế mà đề tài gặp phải. Trong tương lai chúng ta có thể phát triển đề tài này ngày càng hoàn thiện hơn, tối ưu hơn để chúng thực sự trở thành một bộ tính toán số thực dấu phẩy động hoàn hảo nhất, thậm chí là ngang tầm với những bộ FPU đã, đang và sẽ được sử dụng trong đời sống công nghệ 4.0. Cụ thể:

+ FPU được thiết kế bởi với độ chính xác theo tiêu chuẩn IEEE754. Nó không chỉ giải quyết các phép tính cơ bản với số thực dấu phẩy động như: cộng, trừ, nhân, chia mà còn có thể xử lý các hoạt động như dịch, xác định căn bậc và các chức năng siêu việt như sin, cos.

+ Một trong những hướng phát triển của bộ FPU là tăng tốc độ xử lý. Với sự tăng cường của các công nghệ mới như kiến trúc vi xử lý, bộ nhớ RAM và các bộ xử lý đồ họa, bộ FPU có thể được phát triển để tận dụng các công nghệ này và tăng tốc độ xử lý số học.

+ Với các ứng dụng yêu cầu độ chính xác cao như máy tính vật lý, khoa học dữ liệu, hoặc tài chính, cải tiến độ chính xác của bộ FPU sẽ rất quan trọng. Những cải tiến này có thể đạt được thông qua việc tăng số bit của các số thập phân dấu chấm động hoặc sử dụng các kỹ thuật mới để tính toán các số thập phân.

+ Thiết kế và tích hợp thêm bộ chuyển đổi từ số thập phân sang chuẩn biểu diễn số thực dấu phẩy động IEEE 754, khả năng nhập liệu từ các thiết bị ngoại vi như bàn phím máy tính,...

+ Hơn nữa, bộ FPU cũng có thể được phát triển để tương thích với các công nghệ mới như trí tuệ nhân tạo (AI) và học sâu (deep learning). Việc phát triển bộ FPU nhằm tối ưu hóa tính toán số học cho các mô hình AI và học sâu có thể giúp tăng tốc độ và hiệu suất của các ứng dụng này.

+ Ngoài ra, bộ FPU cũng có thể được phát triển để tương thích với các thiết bị di động như smartphone và máy tính bảng. Với sự phổ biến của các ứng dụng di động và tính năng xử lý số học yêu cầu cao, việc phát triển bộ FPU có thể giúp cải thiện hiệu suất và độ chính xác của các ứng dụng di động này.

PHẦN PHỤ LỤC
BẢNG PHÂN CÔNG NHIỆM VỤ

Nội dung thực hiện	Sinh viên thực hiện	Mức độ hoàn thiện
CHƯƠNG 1: TỔNG QUAN		
Nội dung: + Đặt vấn đề, mục tiêu, nội dung, bố cục và giới hạn của đề tài. + Tổng hợp nội dung các thành viên.	Nguyễn Thị Trúc Mai	Tốt
CHƯƠNG 2: CƠ SỞ LÝ THUYẾT		
Nội dung : + Giới thiệu về phương pháp biểu diễn số thực dấu phẩy động. + Tìm hiểu về chuẩn IEEE 754 – chuẩn dấu phẩy động trong máy tính ngày nay.	Nguyễn Quỳnh Đình	Tốt
CHƯƠNG 3: CÁC MODULE PHỤC VỤ VÀ SƠ ĐỒ KHỐI CỦA BỘ FPU		
Nội dung: + Sơ đồ khối của hệ thống + Các module phục vụ cho hệ thống	Phạm Lê Trường Vũ	Tốt
CHƯƠNG 4: THIẾT KẾ VÀ GIẢI THUẬT		
Nội dung: + Thiết kế bộ cộng – trừ + Thiết kế bộ nhân – chia + Tổng hợp các module	Đỗ Trung Hậu	Tốt
CHƯƠNG 5: ĐÁNH GIÁ KẾT QUẢ ĐẠT ĐƯỢC QUA TESTBENCH		
Nội dung: + Mô hình testbench tổng quát + Mô tả các testcase + Kết quả đạt được + Nhận xét	Phan Văn Nguyên	Tốt
CHƯƠNG 6: KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN		
Nội dung: + Đánh giá và nhận xét + Hướng phát triển trong tương lai + Định dạng file, rà soát lỗi chính tả, ngữ pháp, ngữ nghĩa, bố cục,...	Nguyễn Thị Trúc Mai	Tốt

TÀI LIỆU THAM KHẢO

[1] Digital Systems Design Using VHDL Hardcover – March 30, 2007 by Jr. Charles H. Roth (Author), Lizy K. John (Author).

[2] Design of a Floating-Point Fused Add-Subtract Unit Using Verilog Mayank Sharma, Prince Nagar, Ghanshyam Kumar Singh & Ram Mohan Mehra ,Department of Electronics and Communication Engineering School of Engineering & Technology Sharda University, Knowledge Park-III, Greater Noida, (UP), India.

[3] Design of Floating Point Multiplier for Signal Processing Applications - A. Rakesh Babu, R. Saikiran and Sivanantham S. School of Electronics Engineering, VIT UniversityVellore – 632014, Tamilnadu, India.{rakesh40622348,saikiran.18692}@gmail.com , ssivanantham@vit.ac.in

[4] Website, https://github.com/

PHỤ LỤC

CHƯƠNG TRÌNH CHÍNH VÀ TESTBENCH

PHẦN 1: CHƯƠNG TRÌNH CHÍNH

1. Các module phục vụ trong chương trình

```
module Reduction_and8bit(input [7:0] in,output out);
        wire w1,w2,w3,w4,w5,w6;
        and(w1,in[1],in[0]);
        and(w2,in[2],w1);
        and(w3,in[3],w2);
        and(w4,in[4],w3);
        and(w5,in[5],w4);
        and(w6,in[6],w5);
        and(out,in[7],w6);
endmodule

module Reduction_or8bit(input [7:0] in,output out);
        wire w1,w2,w3,w4,w5,w6;
        or(w1,in[1],in[0]);
        or(w2,in[2],w1);
        or(w3,in[3],w2);
        or(w4,in[4],w3);
        or(w5,in[5],w4);
        or(w6,in[6],w5);
        or(out,in[7],w6);
endmodule

module Reduction_or24bit(input [23:0] in,output out);
        Reduction_or8bit RO01(.in(in[7:0]),.out(o1));
        Reduction_or8bit RO02(.in(in[15:8]),.out(o2));
        Reduction_or8bit RO03(.in(in[23:16]),.out(o3));
        or(out,o1,o2,o3);
endmodule

module Reduction_nor31bit(input [30:0] in,output out);
        Reduction_or24bit RO01(.in(in[23:0]),.out(o1));
        Reduction_or8bit RO02(.in({1'b0,in[30:24]}),.out(o2));
        nor(out,o1,o2);
endmodule
```

```
module Complement8bit(input [7:0] in,output [7:0] out);
        not(out[0],in[0]);
        not(out[1],in[1]);
        not(out[2],in[2]);
        not(out[3],in[3]);
        not(out[4],in[4]);
        not(out[5],in[5]);
        not(out[6],in[6]);
        not(out[7],in[7]);
endmodule

module Complement24bit(input [23:0] in,output [23:0] out);
        Complement8bit C01(.in(in[7:0]),.out(out[7:0]));
        Complement8bit C02(.in(in[15:8]),.out(out[15:8]));
        Complement8bit C03(.in(in[23:16]),.out(out[23:16]));
endmodule

module Adder4bit(input [3:0] a,input [3:0] b,input cin,output [3:0]sum,output cout);
        wire g0,g1,g2,g3,p0,p1,p2,p3,c2,c1,c0;
        assign g0 = a[0]&b[0];
        assign g1 = a[1]&b[1];
        assign g2 = a[2]&b[2];
        assign g3 = a[3]&b[3];
        assign p0 = a[0]^b[0];
        assign p1 = a[1]^b[1];
        assign p2 = a[2]^b[2];
        assign p3 = a[3]^b[3];
        assign c0 = g0 |( p0 & cin);
        assign c1 = g1 | (p1&g0)| (p1&p0&cin);
        assign c2 = g2 | (p2&g1) | (p2&p1&g0) | (p2&p1&p0&cin);
        assign cout = g3 | (p3&g2) | (p3&p2&g1) | (p3&p2&p1&g0) | (p3&p2&p1&p0&cin);

        xor(sum[0],p0,cin);
        xor(sum[1],p1,c0);
        xor(sum[2],p2,c1);
        xor(sum[3],p3,c2);

endmodule

module Adder8bit(input [7:0] a,input [7:0] b,input cin,output [7:0]sum,output cout);
        Adder4bit ADD01(.a(a[3:0]),.b(b[3:0]),.cin(cin),.sum(sum[3:0]),.cout(ctemp));
        Adder4bit ADD02(.a(a[7:4]),.b(b[7:4]),.cin(ctemp),.sum(sum[7:4]),.cout(cout));
endmodule
```

```
module Adder9bit(input [8:0] a,input [8:0] b,input cin,output [8:0]sum,output cout);
        Adder8bit ADD01(.a(a[7:0]),.b(b[7:0]),.cin(cin),.sum(sum[7:0]),.cout(ctemp));
        xor(sum[8],a[8],b[8],ctemp);
        assign cout = a[8]&b[8] | a[8]&ctemp | ctemp&b[8];
endmodule

module Adder24bit(input [23:0] a,input [23:0] b,input cin,output [23:0]sum,output cout);
        Adder8bit ADD01(.a(a[7:0]),.b(b[7:0]),.cin(cin),.sum(sum[7:0]),.cout(ctemp1));
        Adder8bit ADD02(.a(a[15:8]),.b(b[15:8]),.cin(ctemp1),.sum(sum[15:8]),.cout(ctemp2));
        Adder8bit ADD03(.a(a[23:16]),.b(b[23:16]),.cin(ctemp2),.sum(sum[23:16]),.cout(cout));
endmodule

module Complement8bit_2s(input [7:0] in,output [7:0] out);
        wire [7:0] outtemp;
        Complement8bit C01(.in(in),.out(outtemp));
        Adder8bit ADD01(.a(outtemp),.b(8'b0000_0001),.cin(1'b0),.sum(out),.cout());
endmodule

module Complement24bit_2s(input [23:0] in,output [23:0] out);
        wire [23:0] outtemp;
        Complement24bit C01(.in(in),.out(outtemp));
        Adder24bit ADD01(.a(outtemp),.b(24'b0000_0000_0000_0000_0000_0001),.cin(1'b0),.sum(out),.cout());
endmodule

module Mux_1Bit(input in0,input in1 ,input sl,output out);
        wire w1,w2,invSL;
        not(invSL,sl);
        and(w1,in0,invSL);
        and(w2,in1,sl);
        or(out,w1,w2);
endmodule
module Mux_8Bit(input [7:0] in0,input [7:0] in1 ,input sl,output [7:0] out);
        Mux_1Bit M01(.in0(in0[0]),.in1(in1[0]) ,.sl(sl),.out(out[0]));
        Mux_1Bit M02(.in0(in0[1]),.in1(in1[1]) ,.sl(sl),.out(out[1]));
        Mux_1Bit M03(.in0(in0[2]),.in1(in1[2]) ,.sl(sl),.out(out[2]));
        Mux_1Bit M04(.in0(in0[3]),.in1(in1[3]) ,.sl(sl),.out(out[3]));
        Mux_1Bit M05(.in0(in0[4]),.in1(in1[4]) ,.sl(sl),.out(out[4]));
        Mux_1Bit M06(.in0(in0[5]),.in1(in1[5]) ,.sl(sl),.out(out[5]));
        Mux_1Bit M07(.in0(in0[6]),.in1(in1[6]) ,.sl(sl),.out(out[6]));
        Mux_1Bit M08(.in0(in0[7]),.in1(in1[7]) ,.sl(sl),.out(out[7]));
endmodule

module Mux_24Bit(input [23:0] in0,input [23:0] in1 ,input sl,output [23:0] out);
        Mux_8Bit M01(.in0(in0[7:0]),.in1(in1[7:0]) ,.sl(sl),.out(out[7:0]));
        Mux_8Bit M02(.in0(in0[15:8]),.in1(in1[15:8]) ,.sl(sl),.out(out[15:8]));
        Mux_8Bit M03(.in0(in0[23:16]),.in1(in1[23:16]) ,.sl(sl),.out(out[23:16]));
endmodule
```

```
module Mux_32Bit(input [31:0] in0,input [31:0] in1 ,input sl,output [31:0] out);
        Mux_24Bit M01(.in0(in0[23:0]),.in1(in1[23:0]),.sl(sl),.out(out[23:0]));
        Mux_8Bit M02(.in0(in0[31:24]),.in1(in1[31:24]),.sl(sl),.out(out[31:24]));
endmodule

module Multiplier24bit(input [23:0] a,input [23:0] b,output [47:0]mul);
        assign mul = a*b;
endmodule

module Divider24bit(input [47:0] a,input [23:0] b,output [24:0]div);
        wire [47:0] div_temp;
        assign div_temp = a/b;
        assign div = div_temp[24:0];
endmodule

module normalizeMandfindShift(
                                        input[23:0] M_result,
                                        input M_carry,
                                        input real_oper,
                                        output reg [22:0] normalized_M,
                                        output reg [4:0] shift
                                        );

reg [23:0] M_temp;

always @(*)
begin
        if(M_carry & !real_oper)
        begin
                normalized_M = M_result[23:1] + {22'b0,M_result[0]};
                shift = 5'd0;
        end
        else

begin
        casex(M_result)
                24'b1xxx_xxxx_xxxx_xxxx_xxxx_xxxx:
                begin
                        normalized_M = M_result[22:0];
                        shift = 5'd0;
                end
                24'b01xx_xxxx_xxxx_xxxx_xxxx_xxxx:
                begin
                        M_temp = M_result << 1;
                        normalized_M = M_temp[22:0];
                        shift = 5'd1;
                end
```

```
24'b001x_xxxx_xxxx_xxxx_xxxx_xxxx:
begin
        M_temp = M_result << 2;
        normalized_M = M_temp[22:0];
        shift = 5'd2;
end
24'b0001_xxxx_xxxx_xxxx_xxxx_xxxx:
begin
        M_temp = M_result << 3;
        normalized_M = M_temp[22:0];
        shift = 5'd3;
end
24'b0000_1xxx_xxxx_xxxx_xxxx_xxxx:
begin
        M_temp = M_result << 4;
        normalized_M = M_temp[22:0];
        shift = 5'd4;
end
24'b0000_01xx_xxxx_xxxx_xxxx_xxxx:
begin
        M_temp = M_result << 5;
        normalized_M = M_temp[22:0];
        shift = 5'd5;
end
24'b0000_001x_xxxx_xxxx_xxxx_xxxx:
begin
        M_temp = M_result << 6;
        normalized_M = M_temp[22:0];
        shift = 5'd6;
end
24'b0000_0001_xxxx_xxxx_xxxx_xxxx:
begin
        M_temp = M_result << 7;
        normalized_M = M_temp[22:0];
        shift = 5'd7;
end
24'b0000_0000_1xxx_xxxx_xxxx_xxxx:
begin
        M_temp = M_result << 8;
        normalized_M = M_temp[22:0];
        shift = 5'd8;
end
24'b0000_0000_01xx_xxxx_xxxx_xxxx:
begin
        M_temp = M_result << 9;
        normalized_M = M_temp[22:0];
        shift = 5'd9;
end
```

```
24'b0000_0000_001x_xxxx_xxxx_xxxx:
begin
        M_temp = M_result << 10;
        normalized_M = M_temp[22:0];
        shift = 5'd10;
end
24'b0000_0000_0001_xxxx_xxxx_xxxx:
begin
        M_temp = M_result << 11;
        normalized_M = M_temp[22:0];
        shift = 5'd11;
end
24'b0000_0000_0000_1xxx_xxxx_xxxx:
begin
        M_temp = M_result << 12;
        normalized_M = M_temp[22:0];
        shift = 5'd12;
end
24'b0000_0000_0000_01xx_xxxx_xxxx:
begin
        M_temp = M_result << 13;
        normalized_M = M_temp[22:0];
        shift = 5'd13;
end
24'b0000_0000_0000_001x_xxxx_xxxx:
begin
        M_temp = M_result << 14;
        normalized_M = M_temp[22:0];
        shift = 5'd14;
end
24'b0000_0000_0000_0001_xxxx_xxxx:
begin
        M_temp = M_result << 15;
        normalized_M = M_temp[22:0];
        shift = 5'd15;
end
24'b0000_0000_0000_0000_1xxx_xxxx:
begin
        M_temp = M_result << 16;
        normalized_M = M_temp[22:0];
        shift = 5'd16;
end
24'b0000_0000_0000_0000_01xx_xxxx:
begin
        M_temp = M_result << 17;
        normalized_M = M_temp[22:0];
        shift = 5'd17;
end
```

```
                    24'b0000_0000_0000_0000_001x_xxxx:
                    begin
                            M_temp = M_result << 18;
                            normalized_M = M_temp[22:0];
                            shift = 5'd18;
                    end
                    24'b0000_0000_0000_0001_0001_xxxx:
                    begin
                            M_temp = M_result << 19;
                            normalized_M = M_temp[22:0];
                            shift = 5'd19;
                    end
                    24'b0000_0000_0000_0000_0000_1xxx:
                    begin
                            M_temp = M_result << 20;
                            normalized_M = M_temp[22:0];
                            shift = 5'd20;
                    end
                    24'b0000_0000_0000_0000_0000_01xx:
                    begin
                            M_temp = M_result << 21;
                            normalized_M = M_temp[22:0];
                            shift = 5'd21;
                    end
                    24'b0000_0000_0000_0000_0000_001x:
                    begin
                            M_temp = M_result << 22;
                            normalized_M = M_temp[22:0];
                            shift = 5'd22;
                    end

                    end
                    24'b0000_0000_0000_0000_0000_0001:
                    begin
                            M_temp = M_result << 23;
                            normalized_M = M_temp[22:0];
                            shift = 5'd23;
                    end
                    default:
                    begin
                            normalized_M = 23'b0;
                            shift = 5'd0;
                    end
            endcase
        end
end

endmodule
```

2. MODULE CỘNG – TRỪ

```
module add_sub(
    input [31:0] n1,
    input [31:0] n2,
    output [31:0] result,
    input sub,
    output Overflow,
    output Underflow,
    output Exception
    );

wire real_oper,real_sign,M_carry;
wire isE1LessThanE2,reduced_and_E1,reduced_and_E2,reduced_or_E1,reduced_or_E2;
wire [7:0] temp_exp_diff,One_Added_E,new_E,complemented_temp_exp_diff,exp_diff,E,complemented_E2,complemented_shift_E;
wire [8:0] final_E;
wire [23:0] M1,M2,complemented_M2,complemented_M_result,M_result,M_result2,new_M2;
wire w1,w2,w3,final_sign;
wire [22:0] final_M;
wire[4:0] shift_E;

// If the bits of E1, E2 are 1 ==> Then the number will be either infinity or NAN ( i.e. an Exception )
Reduction_and8bit RA01(.in(n1[30:23]),.out(reduced_and_E1));
Reduction_and8bit RA02(.in(n2[30:23]),.out(reduced_and_E2));

// If any of E1 or E2 has all btis 1 then we have an Exception( high )
or(Exception,reduced_and_E1,reduced_and_E2);

// If all the bits of E1 or E2 are 0  ===> Number is denormalized and implied bit of the corresponding mantissa is set as 0.
Reduction_or8bit RO01(.in(n1[30:23]),.out(reduced_or_E1));
Reduction_or8bit RO02(.in(n1[30:23]),.out(reduced_or_E2));

// Performing E1 - E2
// Before subtraction, complementing E2 bcoz of 2's complement subtraction
Complement8bit C01(.in(n2[30:23]),.out(complemented_E2));
Adder8bit ADD01(.a(n1[30:23]),.b(complemented_E2),.cin(1'b1),.sum(temp_exp_diff),.cout(isE1GreaterThanE2));

// If exp_diff comes out to be -ve ===> Found it's 2's complement
// Original or 2's complement version is selected according to isE1GreaterThanE2
Complement8bit_2s C023(.in(temp_exp_diff),.out(complemented_temp_exp_diff));
Mux_8Bit M011(.in0(complemented_temp_exp_diff),.in1(temp_exp_diff),.sl(isE1GreaterThanE2),.out(exp_diff));

// Selecting the larger exponent
Mux_8Bit M03(.in0(n2[30:23]),.in1(n1[30:23]),.sl(isE1GreaterThanE2),.out(E));

// shifting either mantissa of n1 or n2 a/c to isE1GreaterThanE2
assign M1 = isE1GreaterThanE2? {reduced_or_E1,n1[22:0]}:{reduced_or_E1,n1[22:0]} >> exp_diff;
assign M2 = isE1GreaterThanE2?{reduced_or_E2,n2[22:0]} >> exp_diff:{reduced_or_E2,n2[22:0]};
```

```
// assuming real_oper and real_sign
xor(real_oper,sub,n1[31],n2[31]);
buf(real_sign,n1[31]);

// M2 is added to or subtracted from M1 a/c to real_oper
Complement24bit C02(.in(M2),.out(complemented_M2));
Mux_24Bit M04(.in0(M2),.in1(complemented_M2),.sl(real_oper),.out(new_M2));
Adder24bit ADD02(.a(M1),.b(new_M2),.cin(real_oper),.sum(M_result),.cout(M_carry));

// correction in the sign of the final result
and(w1,~real_sign,real_oper,~M_carry);
and(w2,~real_oper,real_sign);
and(w3,M_carry,real_sign);
or(final_sign,w1,w2,w3);

// 1 is added to E if Addtion is performed b/w mantissae and carry is generated
Adder8bit ADD0212(.a(E),.b(8'd1),.cin(1'b0),.sum(One_Added_E),.cout());
Mux_8Bit M031(.in0(E),.in1(One_Added_E),.sl(M_carry&!real_oper),.out(new_E));

// if M_result is negative then 2's complement of M_result is to be calculated
Complement24bit_2s C03(.in(M_result),.out(complemented_M_result));
Mux_24Bit M05(.in0(M_result),.in1(complemented_M_result),.sl(real_oper&!M_carry),.out(M_result2));

// Normalization step ( See Utils.v )
normalizeMandfindShift NM(.M_result(M_result2),.M_carry(M_carry),.real_oper(real_oper),.normalized_M(final_M),.shift(shift_E));
Complement8bit C04(.in({3'b000,shift_E}),.out(complemented_shift_E));

// finally shift is subtracted from E ( 2's complement subtraction )
Adder8bit ADD03(.a(new_E),.b(complemented_shift_E),.cin(1'b1),.sum(final_E[7:0]),.cout(final_E[8]));

// final ans
assign result = {final_sign,final_E[7:0],final_M};

// if (Carry) final_E[8] = 0 ===> final_E is -ve ( Underflow )
not(Underflow,final_E[8]);

// if All bits of of One_Added_E are 1 ( 255 ) and shift_E are 0 ( 0 ), then final_E is 255 ( Out of bound,i.e, Overflow )
and(Overflow,&One_Added_E,~|shift_E);

endmodule
```

3. MODULE NHÂN

```
module mul(
                                input [31:0] n1,
                                input [31:0] n2,
                                output [31:0] result,
                                output Overflow,
                                output Underflow,
                                output Exception
                            );

wire [8:0] sum_E,final_E;
wire [47:0] M_mul_result;
wire [23:0] normalized_M_mul_result;
wire [22:0] final_M;
wire final_sign,reduced_and_E1,reduced_and_E2,reduced_or_E1,reduced_or_E2,carry_E;
```

```
// Checking whether all the bits of E1, E2 are 1 ==> Then the number will be either infinity or NAN ( i.e. an Exception )
Reduction_and8bit RA01(.in(n1[30:23]),.out(reduced_and_E1));
Reduction_and8bit RA02(.in(n2[30:23]),.out(reduced_and_E2));

// If any of E1 or E2 has all btis 1 then we have an Exception( high )
or(Exception,reduced_and_E1,reduced_and_E2);

// final sign of the result
xor(final_sign,n1[31],n2[31]);

// if all the bits of E1 or E2 are 0  ===> Number is denormalized and implied bit of the corresponding mantissa is set as 0.
Reduction_or8bit RO01(.in(n1[30:23]),.out(reduced_or_E1));
Reduction_or8bit RO02(.in(n1[30:23]),.out(reduced_or_E2));

// Multiplying M1 and M2 ( here we have firstly concatenate the implied bit with the corresponding mantissa )
Multiplier24bit MUL01(.a({reduced_or_E1,n1[22:0]}),.b({reduced_or_E2,n2[22:0]}),.mul(M_mul_result));

// MSB of the product is used as select line
// finding the rounding bit ( finally we will or with the LSB of the final product to include rounding )
// if M_mul_result[47] is 1 ===> product is normalized and we will round off the last 24 bits else last 23 bits
Reduction_or24bit RO03(.in({1'b0,M_mul_result[22:0]}),.out(mul_round1));
Reduction_or24bit RO04(.in(M_mul_result[23:0]),.out(mul_round2));
Mux_1Bit M01(.in0(mul_round1),.in1(mul_round2),.sl(M_mul_result[47]),.out(final_product_round));

// normalization
// if MSB of M_mul_result is 1 ===> product is already normalized and next 23 bits after MSB is taken
// if MSB of M_mul_result is 0 ===> The next bit is always 1, so starting from next to next bit, next 23 bits are taken
// here we do not require to shift any bit
Mux_24Bit M02(.in0({1'b0,M_mul_result[45:23]}),.in1({1'b0,M_mul_result[46:24]}),.sl(M_mul_result[47]),.out(normalized_M_mul_result));

Adder24bit ADD23(.a({1'b0,normalized_M_mul_result[22:0]}),.b({23'b0,final_product_round}),.cin(1'b0),.sum({temp,final_M}),.cout());

// Adding E1 and E2
Adder8bit ADD01(.a(n1[30:23]),.b(n2[30:23]),.cin(1'b0),.sum(sum_E[7:0]),.cout(sum_E[8]));

// Subtracting 127(BIAS) from sum_E = E1 + E2
// if M_mul_result[47] = 1 ===> product is of the form 11.(something) and we need to shift the decimal point to left to make the product normalized and therefore we add 1 to r
// if M_mul_result[47] = 0 ===> product is of the form 01.(something) and the product is already normalized and nothing is added or subtracted to E
Adder9bit ADD02(.a(sum_E),.b(9'b110000001),.cin(M_mul_result[47]),.sum(final_E),.cout(carry_E));

// In 2's complement subtraction :
// if carry_E = 0 ===> result is negative and it the case of Underflow
// if carry_E = 1 and MSB of sum(final_E) is 8 (that means sum is atleast 256 ) ===> it is the case of Overflow
not(Underflow,carry_E);
and(Overflow,carry_E,final_E[8]);

assign result = {final_sign,final_E[7:0],final_M};

endmodule
```

4. MODULE CHIA

```
module div(
                        input [31:0] n1,
                        input [31:0] n2,
                        output [31:0] result,
                        output Overflow,
                        output Underflow,
                        output Exception
            );
```

```
wire is_n2_zero,reduced_and_E1,reduced_and_E2,reduced_or_E1,reduced_or_E2,Overflow1,Underflow1,Overflow2,Underflow2;
wire [24:0] M_div_result;
wire [8:0] temp_E2,temp_E3;
wire [7:0] complemented_E2,complemented_shift_E1,sub_E,bias_added_E,temp_E1,final_E;
wire [4:0] shift_E1,shift_E2;
wire [22:0] normalized_M1,normalized_M2,final_M;

//if all the bits of E1 or E2 are 1 or if n2 is zero ===> Exception
Reduction_and8bit RA01(.in(n1[30:23]),.out(reduced_and_E1));
Reduction_and8bit RA02(.in(n2[30:23]),.out(reduced_and_E2));
Reduction_nor31bit RN01(.in(n2[30:0]),.out(is_n2_zero));
or(Exception,reduced_and_E1,reduced_and_E2,is_n2_zero);

// final sign of the result
xor(final_sign,n1[31],n2[31]);

// if all the bits of E1 or E2 are 0  ===> Number is denormalized and the implied bit of the corresponding mantissa is to be set as 0.
Reduction_or8bit RO01(.in(n1[30:23]),.out(reduced_or_E1));
Reduction_or8bit RO02(.in(n1[30:23]),.out(reduced_or_E2));

// Subtracting E2 from E1 ===> 2's complement Subtraction
Complement8bit C01(.in(n2[30:23]),.out(complemented_E2));
Adder8bit ADD01(.a(n1[30:23]),.b(complemented_E2),.cin(1'b1),.sum(sub_E),.cout());

// Adding 127(BIAS) to sub_E
Adder8bit ADD02(.a(sub_E),.b(8'b01111111),.cin(1'b0),.sum(bias_added_E),.cout());

// Used to make all mantissae normalized if any of the them is firstly denormalized
normalizeMandfindShift NM1(.M_result({reduced_or_E1,n1[22:0]}),.M_carry(1'b0),.real_oper(1'b0),.normalized_M(normalized_M1),.shift(shift_E1));
normalizeMandfindShift NM2(.M_result({reduced_or_E2,n2[22:0]}),.M_carry(1'b0),.real_oper(1'b0),.normalized_M(normalized_M2),.shift(shift_E2));

// dividing M1 by M2
Divider24bit DIV01(.a({1'b1,normalized_M1,24'b0}),.b({1'b1,normalized_M2}),.div(M_div_result));

// if M_div_result[24] = 0 ===> take ans from 22 pos to 0 pos, i.e, final_M = M_div_result[22:0]
// if M_div_result[24] = 1 ===> take ans from 23 pos to 1 pos, i.e, final_M = M_div_result[23:1]
Mux_24Bit M02(.in0({1'b0,M_div_result[22:0]}),.in1({1'b0,M_div_result[23:1]}),.sl(M_div_result[24]),.out({temp,final_M}));
  // Subtracting shift_E1 from bias_added_E  ===> we get temp_E1
  Complement8bit C02(.in({3'b000,shift_E1}),.out(complemented_shift_E1));
  Adder8bit ADD03(.a(bias_added_E),.b(complemented_shift_E1),.cin(1'b1),.sum(temp_E1),.cout());

  // Adding shift_E2 to temp_E1 ===> we get temp_E2
  Adder8bit ADD04(.a(temp_E1),.b({3'b000,shift_E2}),.cin(1'b0),.sum(temp_E2[7:0]),.cout(temp_E2[8]));
  and(Overflow1,temp_E1[8],temp_E2[8]);
  nor(Underflow1,temp_E1[8],temp_E2[8]);

  // Subtracting 1 from temp_E2[7:0] to get temp_E3
  Adder8bit ADD05(.a(temp_E2[7:0]),.b(8'b11111111),.cin(1'b0),.sum(temp_E3[7:0]),.cout(temp_E3[8]));
  and(Overflow2,temp_E2[8],temp_E3[8]);
  nor(Underflow2,temp_E2[8],temp_E3[8]);

  // Based on M_div_result[24] bit ===> we will select temp_E2 or temp_E3
  Mux_8Bit M03(.in0(temp_E3[7:0]),.in1(temp_E2[7:0]),.sl(M_div_result[24]),.out(final_E));
  Mux_1Bit M04(.in0(Overflow2),.in1(Overflow1),.sl(M_div_result[24]),.out(Overflow));
  Mux_1Bit M05(.in0(Underflow2),.in1(Underflow1),.sl(M_div_result[24]),.out(Underflow));

  assign result = {final_sign,final_E[7:0],final_M};

  endmodule
```

5. MODULE TỔNG HỢP

```
module Main(n1,n2,oper,result,Overflow,Underflow,Exception);

input [31:0] n1,n2;
input[1:0] oper;
output Overflow,Underflow,Exception;
output [31:0] result;
wire [31:0] temp_result,result1,result2,result3,result4,result5,result6;
wire Overflow1,Underflow1,Exception1,Overflow2,Underflow2,Exception2,Overflow3,Underflow3,Exception3;

add_sub AS(.n1(n1),.n2(n2),.result(result1),.sub(|oper),.Overflow(Overflow1),.Underflow(Underflow1),.Exception(Exception1));
mul M(.n1(n1),.n2(n2),.result(result2),.Overflow(Overflow2),.Underflow(Underflow2),.Exception(Exception2));
div D(.n1(n1),.n2(n2),.result(result3),.Overflow(Overflow3),.Underflow(Underflow3),.Exception(Exception3));

Mux_32Bit M01(.in0(result1),.in1(result2),.sl(oper[1]),.out(temp_result));
Mux_32Bit M02(.in0(temp_result),.in1(result3),.sl(&oper),.out(result4));

Mux_1Bit M03(.in0(Overflow1),.in1(Overflow2),.sl(oper[1]),.out(temp1));
Mux_1Bit M04(.in0(temp1),.in1(Overflow3),.sl(&oper),.out(Overflow));

Mux_1Bit M05(.in0(Underflow1),.in1(Underflow2),.sl(oper[1]),.out(temp2));
Mux_1Bit M06(.in0(temp2),.in1(Underflow3),.sl(&oper),.out(Underflow));

Mux_1Bit M07(.in0(Exception1),.in1(Exception2),.sl(oper[1]),.out(temp3));
Mux_1Bit M08(.in0(temp3),.in1(Exception3),.sl(&oper),.out(Exception));

// if Exception is 1 ===> set the result to all 1s
Mux_32Bit M09(.in0(result4),.in1(32'b1_11111111_11111111111111111111111),.sl(Exception),.out(result5));

// if Underflow is 1 ===> set the result to all 0s and sign is the final_sign ( setting to 0 )
Mux_32Bit M010(.in0(result5),.in1({result4[31],31'b00000000_00000000000000000000000}),.sl(Underflow),.out(result6));

// if Overflow is 1 ===> set the E to all 1s and M to all 0s and sign is the final_sign ( setting to +inf or -inf)
Mux_32Bit M011(.in0(result6),.in1({result4[31],31'b11111111_00000000000000000000000}),.sl(Overflow),.out(result));

  endmodule
```

PHẦN 2: TESTBENCH FILE

```
module Main_tb;

        // Inputs
        reg [31:0] n1;
        reg [31:0] n2;
        reg [1:0] oper;

        // Outputs
        wire [31:0] result;
        wire Overflow;
        wire Underflow;
        wire Exception;
```

```
        // Instantiate the Unit Under Test (UUT)
        Main uut (
                .n1(n1),
                .n2(n2),
                .oper(oper),
                .result(result),
                .Overflow(Overflow),
                .Underflow(Underflow),
                .Exception(Exception)
        );

        initial begin
                // Initialize Inputs
                n1 = 32'b01000011000011111000111101011100;    // 143.56
                n2 = 32'b11000010101011101101111110111110;    // -87.437


                oper = 2'd0; #50;

                $display("Addtion result : %b",result);
                $display("Overflow : %b , Underflow : %b , Exception : %b",Overflow,Underflow,Exception);

                oper = 2'd1; #50;

                $display("Subtraction result : %b",result);
                $display("Overflow : %b , Underflow : %b , Exception : %b",Overflow,Underflow,Exception);

                oper = 2'd2; #50;

                $display("Multiplication result : %b",result);
                $display("Overflow : %b , Underflow : %b , Exception : %b",Overflow,Underflow,Exception);

                oper = 2'd3; #50;

                $display("Division result : %b",result);

               $display("Overflow : %b , Underflow : %b , Exception : %b",Overflow,Underflow,Exception);



        end

endmodule
```