**ĐẠI HỌC BÁCH KHOA THÀNH PHỐ HỒ CHÍ MINH**

**Khoa Điện-Điện tử**

**Bộ môn Điện tử**

**

**KỸ THUẬT SỐ NÂNG CAO**

**BÁO CÁO LAB – PHẦN 2:**

**THIẾT KẾ MỘT MÁY TÍNH DẤY CHẤM ĐỘNG THỰC  
HIỆN CÁC PHÉP TOÁN (+, -, \*, /) GIỮA HAI SỐ FLOATING POINT (IEEE754, SINGLE PRECISION, 32-BIT)**

**GVHD: GV. Trịnh Vũ Đăng Nguyên**

**SINH VIÊN: Phạm Xuân Thi - 1814120**

**Nguyễn Nhật Thắng - 1814099**

*Tp. Hồ Chí Minh, 16 tháng 6 năm 2021*

1. **Mục tiêu:**

* Thiết kế một máy tính dấu chấm động thực hiện 4 phép toán (+, -, \*, /) giữa 2 số floating point (IEEE-754, SINGLE PRECISION, 32 BIT).
* Với các ngõ vào, ra như sau:
* Ngõ vào là 2 số floating point 32-bits (A, B) và 2 bits lựa chọn các phép toán (1 trong 4 phép toán trên).
* Ngõ ra là 1 số floating point 32 bits
* Kết quả phép tính ở định dạng IEEE-754.
* Nhóm thực hiện bài báo cáo thực hiện trên ngôn ngữ Verilog, sử dụng mô phỏng trên phần mềm ModelSim – INTEL FPGA STARTER EDITION 10.6c.

1. **Nguyên lý hoạt động của mạch:**
2. **Số Floating point (IEEE-754, SINGLE PRECISION, 32 BIT):**
3. Tại sao chúng ta cần số dấu chấm động?

* Vì bộ nhớ máy tính có giới hạn, chúng ta không thể lưu trữ 1 con số với độ chính xác vô hạn bất chấp chúng ta dùng kiểu nhị phân hay thập phân. Tại một mức nào đó thì ta cần phải cắt giảm bớt. Nhưng độ chính xác đến bao nhiêu là đủ? Bao nhiêu chữ số nguyên và bao nhiêu chữ số phần thực?
* Với một công nhân xây dựng, 10 met hay 10.0001 met là như nhau, phép tính của anh ta không quan tâm đến độ chính xác quá nhỏ như vậy.
* Nhưng với 1 người thiết kế microchip, 0.0001 met đã là 1 sự khác biệt rất lớn.
* Trong vật lý người ta cần tốc độ ánh sáng (3\*109) và hằng số hấp dẫn newton (0.0000000000667) để tính toán cùng nhau.
* Có thể thấy để đáp ứng nhu cầu tính toán đa dạng và cả yêu cầu độ chính xác người ta đưa ra số dấu chấm động.
* Theo chuẩn IEEE-754, số Floating point được chia làm 3 phần:
* Phần dấu (bit 31): với quy ước 0 là số dương, 1 là số âm (sign).
* Phần mũ (từ bit 30 đến bit 23): biễu diễn phần mũ của 1 số (exponent).
* Phần định trị (từ bit 22 đến bit 0): biểu diễn phần sau dấu phẩy của 1 số (fraction).
* Ta làm ví dụ chuyển 1 số thực 32-bit (10.7510×100) sang 1 số floating point 32-bit.
* Chuyển số thập phân 10.7510×100 sang số nhị phân 1010.11×20.
* Chuyển hóa số vừa chuyển thành phần nguyên là 1: 1.01011×23.
* Phần mũ: sign = 02.
* Tính toán phần mũ: exponent = 3 + 127 = 13010 = 100000102.
* Phần định trị: fraction = 010110000000000000000002.
* Vậy ở dạng floating point số 10.75 có dạng:
* 0 10000010 01011000000000000000000 (ở dạng số binary).
* 412c0000 (ở dạng số hexadecimal).
* Để thuận tiện trong việc chuyển đổi các dạng số, nhóm sử dụng công cụ có sẵn trên mạng:

*https://www.h-schmidt.net/FloatConverter/IEEE754.html*

1. Những trường hợp đặc biệt:

* Biễu diễn số 0:



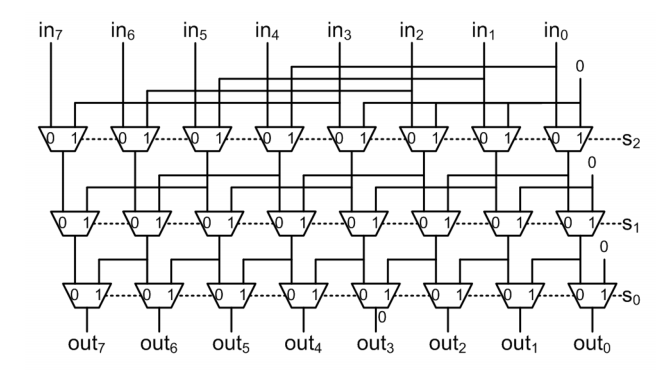
* Trong đó bit S tùy chọn.
* Biểu diễn vô cùng:



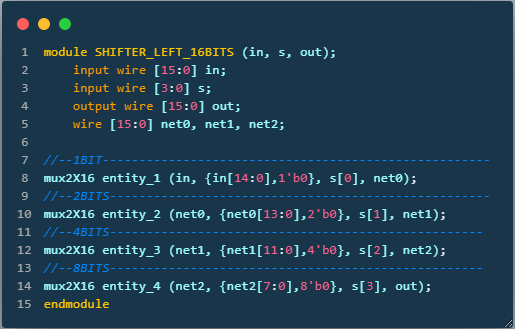
* Trong đó bit S = 0 thể hiện dương vô cùng.
* Trong đó bit S = 1 thể hiện âm vô cùng.
* Biễn diễn NaN (Not a Number):
* Trong trường hợp phần exponent là 128, khi đó số ta có lớn hơn vô cùng, gọi là NaN (Not a Number).
* Kết quả sẽ bị tràn: tràn trên (Overflow) hoặc tràn dưới (Underflow), thì kết quả sẽ trả về NaN nếu kết quả bị tràn trên và trả về 32’h00000000 nếu kết quả bị tràn dưới.

1. **Các bộ quan trọng để thực hiện phép toán:**
2. Bộ dịch bit (SHIFTER):

* Ngõ vào gồm 1 số 2n bits và n bits điều khiển ngõ ra là 1 số 2n bits. Dựa vào n bits điều khiển, giá tri của ngõ ra sẽ là ngõ vào sau khi dịch (sang trái (left) hoặc sang phải (right)) đi 1 số có giá trị n.
* Ví dụ: với bộ dịch trái 8 bits thì ngõ vào bao gồm 1 số 8 bits, 3 bits điều khiển, ngõ ra là 1 số 8 bits. Ta có 3 bits điều khiển: 1012 thì kết quả ngõ ra sẽ được dịch trái 5 bits.
* Bộ dịch bit có mặt trong cả 4 phép toán (+, -, \*, /)
* Sơ đồ khối:

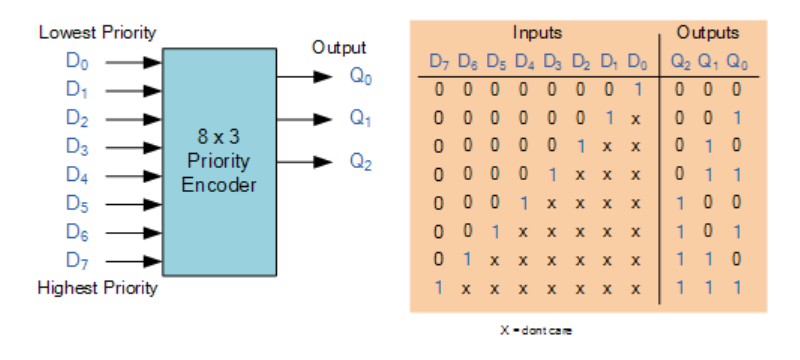


* Dưới đây là đoạn code verilog minh họa bộ dịch 16 bits. Các bộ dich bit lớn hơn như 32 bits, 48 bits, 64 bits thực hiện tương tự.

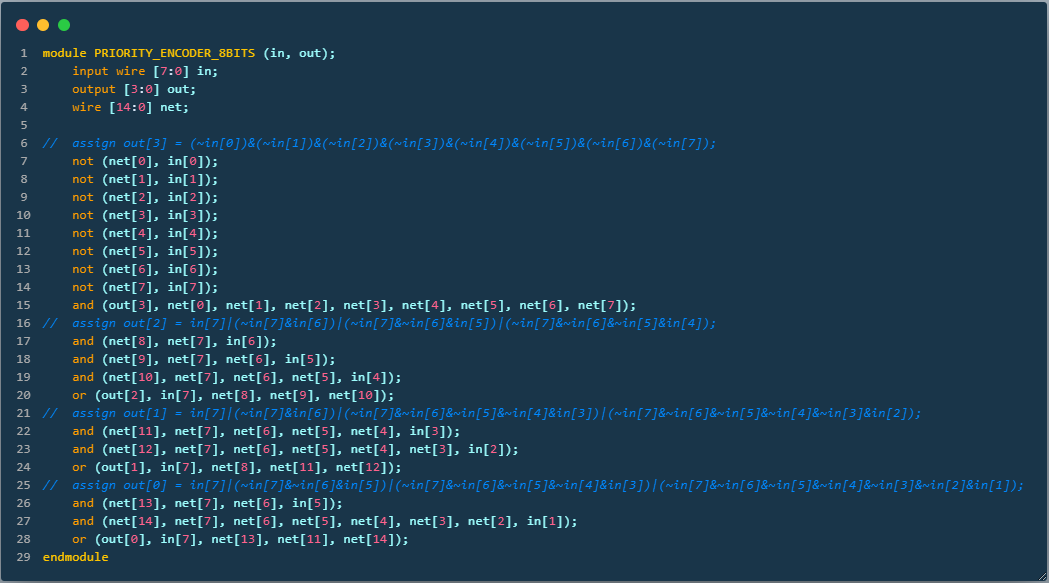
****

1. Bộ phát hiện bit uu tiên (Priority Encoder):

* Ngõ vào gồm 1 số 2n bits, ngõ ra là 1 số n bits thể hiện vị trí của bits 1 đầu tiên:
* Ví dụ: bộ PE 8 bits có ngõ vào 8 bits, ngõ ra 3 bits. Với ngõ vào là 01001011 thì ngõ ra là 110 (bit 1 đầu tiên ở vị trí thứ 6).

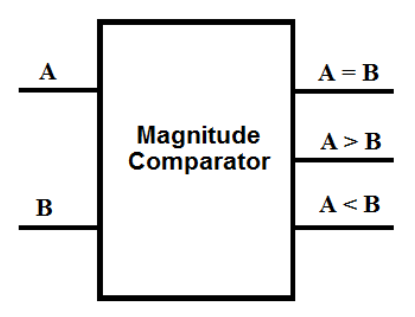


* Dưới đây là đoạn code verilog minh họa bộ dịch 8 bits. Các bộ dich bit lớn hơn như 32 bits, 48 bits, 64 bits thực hiện tương tự.

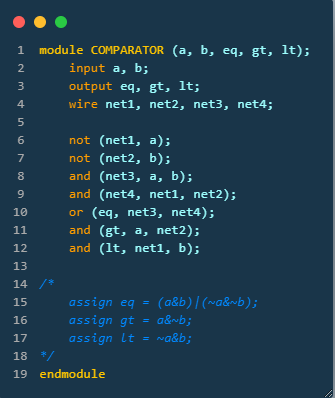


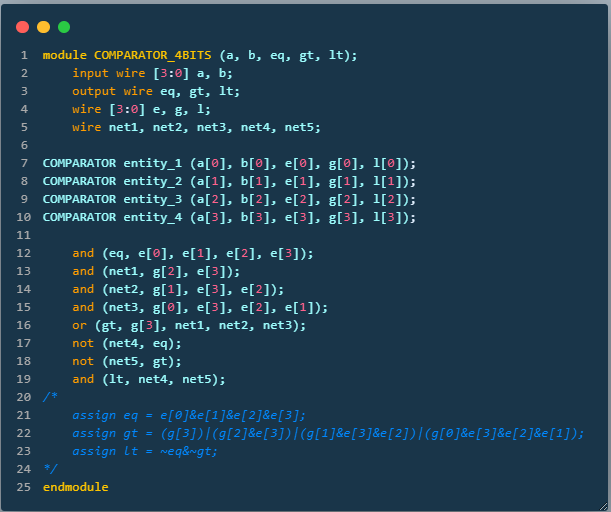
1. Bộ so sánh (Comparator):

* Ngõ vào gồm 2 số 2n bits, ngõ ra gồm 3 số 1 bit: bằng (equal), lớn hơn (greater), nhỏ hơn (less).



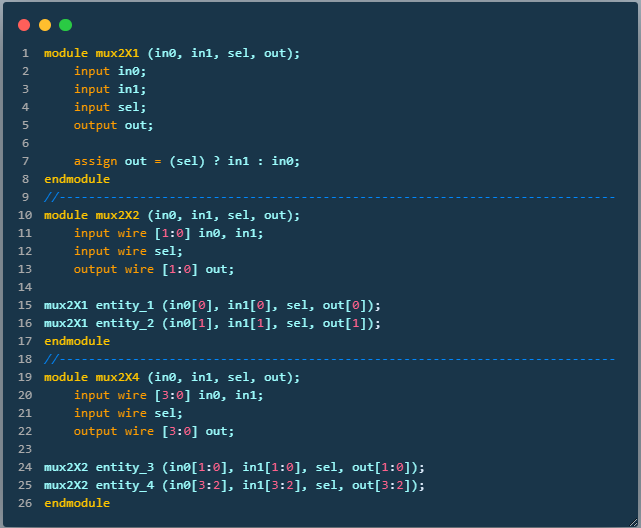
* Dưới đây là đoạn code verilog minh họa bộ dịch 1 bit, 4 bits. Các bộ dich bit lớn hơn như 8 bits, 32 bits, 48 bits, 64 bits thực hiện tương tự.





1. Bộ ghép kênh (MUX):

* Ngõ vào gồm 2 số 2n bits và 1 số 1 bit, ngõ ra là 1 số 2n bits.
* Chức năng: dùng 1bit ngõ vào để chọn ra 1 số trong 2 số ngõ vào 2n bits. Nếu 0 chọn số 2n bits đầu tiên và ngược lại.
* Dưới đây là đoạn code verilog minh họa bộ dịch 1 bit, 2 bits. Các bộ dich bit lớn hơn thực hiện tương tự.



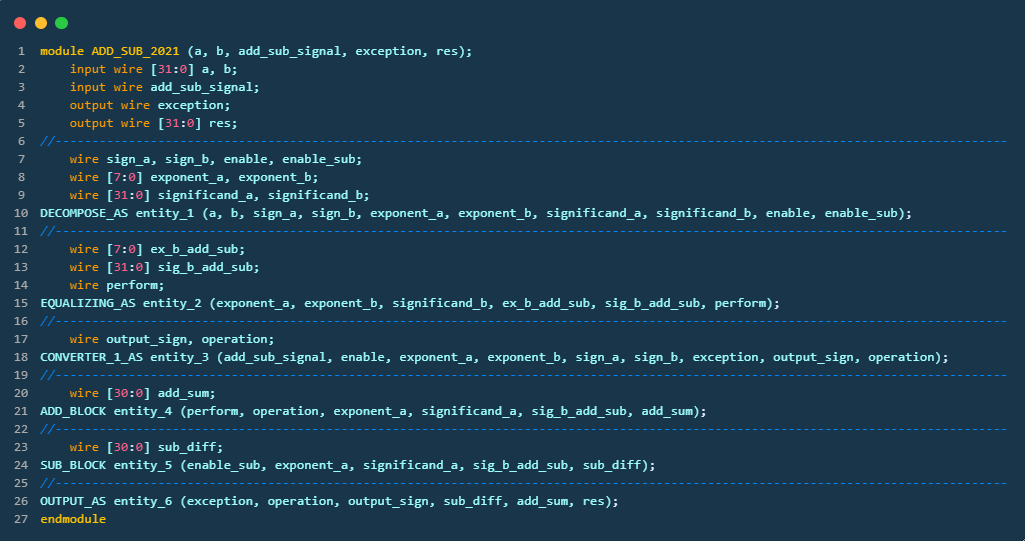
1. **Các phép toán:**
2. **Bộ cộng và bộ trừ:**
3. Giải thuật:

* Chuyển 2 số ngõ vào sang dạng dấu chấm động và tách riêng các phần sign – exponent – significand cho từng số. Ngoài ra cũng mở rộng phần significand thành 32 bits với bit 23 bằng 1, bit 31 đến 24 bằng 0, bit 22 đến 0 lấy từ ngõ vào bit 22 đến 0.
* Ta có 1bit ngõ vào quyết định việc thực hiện phép cộng hay phép trừ: 0 là phép cộng, 1 là phép trừ.
* Đưa phần exponent của 2 số sau khi đã phân tích vào bộ so sánh để tìm ra số có phần exponent nhỏ hơn và phần chênh lệch giữa 2 exponent. Thực hiện dịch phần significand của số có phần exponent nhỏ hơn sang phải tương ứng số phần chênh lệch giữa 2 exponent của 2 số. Sau khi thực hiện, 2 số phải có giá trị giữ nguyên nhưng exponent của 2 số là bằng nhau.
* Sau khi đã hiệu chỉnh 2 số sao cho phần exponent bằng nhau, ta thực hiện phép cộng hoặc trừ.
* Đối với phép cộng: cộng 2 phần significand với nhau (bằng bộ full-adder) và ta thu được phần signìicand mới.
* Đối với phép trừ: trừ 2 phần significand với nhau (bằng bộ full-subtractor); sau đó chuẩn hóa significand lại bằng bộ PE đã được giải thích ở các phép toán quan trọng, để đảm bảo bit 1 đầu tiên nằm ở vị trí bit 23. Số lần cần dịch được dung để hiệu chỉnh phần mũ của kết quả sau khi dịch.
* Tổng hợp lại kết quả ngõ ra thành 1 số 32 bits duy nhất
* Kiểm tra xem kết quả ngõ ra có bị tràn hay không. Nếu tràn thì thông báo qua cờ exception.

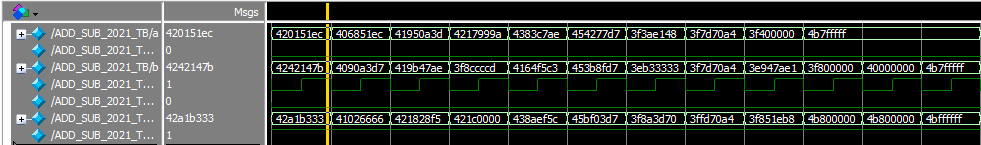
1. Các khối cơ bản: bao gồm 6 khối cơ bản như sau:

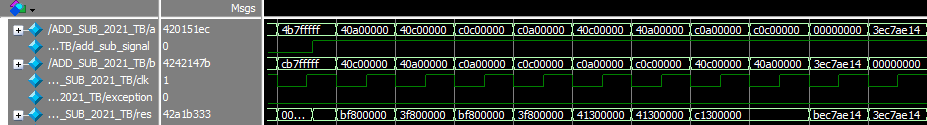
* DECOMPOSE\_AS:
* Có chức năng phân tích 2 số 32 bits đầu vào thành các phần tương ứng bao gồm phần dấu (sign), phần mũ (exponent) và phần định trị (significand).
* Tín hiệu vào bao gồm 2 số ngõ vào 32 bits (a và b). Tín hiệu ngõ ra gồm 6 phần bao gồm phần sign (1 bit), phần exponent (8 bits) và phần significand (32 bits) cho 2 số ngõ vào.
* Ta so sánh 2 số 32 bits ngõ vào, số nào lớn hơn sẽ ra ở ngõ a và nhỏ hơn sẽ ra ở ngõ b. Mục đích chính để cho thuận lợi cho phép cộng trừ.
* Phần significand được mở rộng đã được đề cập ở phần trên.
* EQUALIZING\_AS:
* Có chức năng cân bằng phần exponent và điều chỉnh phần significand để thực hiện phép cộng trừ.
* Tín hiệu vào bao gồm 2 số exponent 8 bits và 1 số significand 32 bits của số nhỏ hơn
* Ngõ ra sẽ là 1 số 32 bits đã được hiệu chỉnh
* Nguyên lý hoạt động: trừ 2 số exponent 8 bits, ta tiến hình dịch phần significand của ngõ vào.
* CONVERTER\_1\_AS:
* Có chức năng xác định bit dấu ngõ ra (output\_sign) và việc thực hiện phép cộng hay phép trừ do bit operation quyết định.
* Nguyên lý hoạt động: việc xác định dấu bit ngõ ra (output\_sign) và 0 hay 1 ta xét đến đang thực hiện phép cộng hay phép trừ và dấu của số của ngõ vào lớn hơn. Xác định bit ngõ ra (operation) thực hiện bằng cách xor dấu của dấu a và b.
* ADD\_BLOCK:
* Có chức năng thực hiện phép toán cộng 32 bits cho phần significand của 2 số. Kết quả trả về là 1 số 32 bits.
* Nếu bit 24 của kết quả trả về là 1 thì phần exponent sẽ được tăng thêm 1 bằng bộ cộng FULL\_ADDER\_8BITS.
* Ngõ ra của bộ này sẽ là 31 bits bao gồm 8 bits từ bit 30 đến 23 của exponent và 23 bits từ bít 23 đến 1 nếu bit kết quả trả về là 1, nếu ngược lại sẽ lấy bit từ 22 đến 0 của kết quả trả về.
* Khối trên được thực hiện bằng bộ cộng FULL\_ADDER\_32BITS.
* SUB\_BLOCK:
* Có chức năng thực hiện phép toán trừ 32 bits cho phần significand của 2 số. Sau đó ta dung bộ PRIORITY\_ENCODER\_32BITS\_SIG được cải tiến từ bộ PE 32 bits đã đề cập ở trên, mục đích để hiệu chỉnh phần exponent.
* Ngõ ra của bộ này sẽ là 31 bits.
* OUTPUT\_AS:
* Có chức năng nên dùng kết quả của bộ ADD\_BLOCK hay kết quả của bộ SUB\_BLOCK theo bit dấu của operation.
* Ngõ ra của bộ này là kết quả phép toán cộng hoặc trừ.

1. Nội dung của file ADD\_SUB\_2021.v:



1. Nội dung khi testbench:



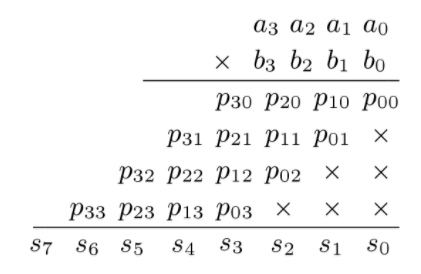


1. **Bộ nhân:**
2. Giải thuật:

* Tương tự như phân tích ở bộ cộng trừ, đầu tiên ta tách 2 số 32 bits của ngõ vào thành 3 thành phần sign, exponent, significand.
* Khi thực hiện nhân 2 số, về cơ bản là thực hiện phép cộng phần exponent, phép nhân cho phần significand và phép xor cho dấu của 2 số.
* Khi thực hiện phép nhân cho phần significand. Về bản chất giống việc thực hiện phép nhân bình thường, ta thực hiện liên tục phép dịch phải và phép cộng hàng dọc xuống. Kết quả phép tính này sẽ có ngõ ra có số bit gấp đôi số bit của ngõ vào. Ví dụ: ta có ngõ vào 2 số 32 bits thì kết quả thu được sẽ là 64 bits của ngõ ra.
* Tổng hợp lại kết quả và kiểm tra xem kết quả ngõ ra có bị tràn trên hay tràn dưới hay không.

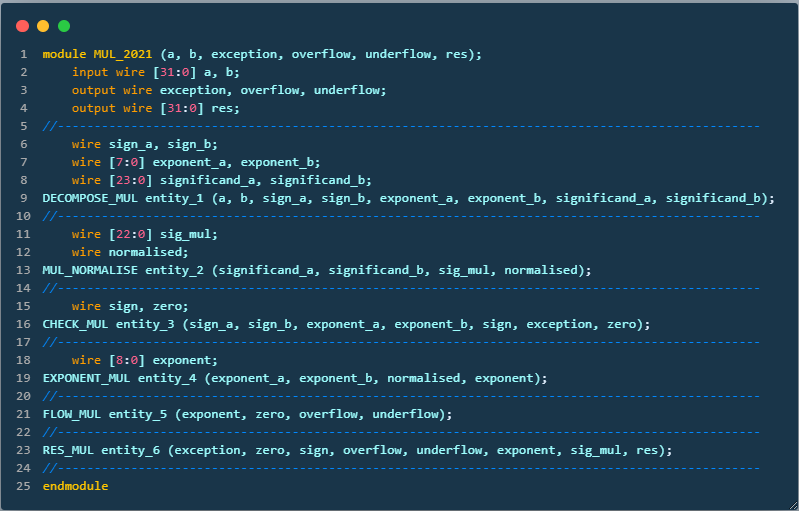
1. Các khối cơ bản: bao gồm 6 khối cơ bản như sau:

* DECOMPOSE\_MUL: có chức năng phân tích 2 số 32 bits ngõ vào thành 3 thành phần bao gồm: sign, exponent, significand. Trong đó phần significand được mở rộng thành 24 bits với bit thứ 23 là bit 1.
* MUL\_NORMALISE: thực hiện đồng thời 2 mục đích
* Thực hiện phép nhân 2 phần significand được 1 số 48 bits. Nguyên lý hoạt động của bộ MULTIPLY bắt nguồn từ phép nhân căn bản. Thực hiện đồng thời phép dịch và bộ cộng nối tiếp.

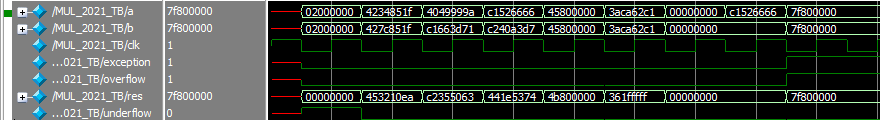


* Thực hiện việc chuẩn hóa kết quả sao cho bits thứ 47 là 1. Kết quả của ngõ ra sẽ là 22 bits đã chuẩn hóa của phần significand.
* CHECK\_MUL: có chức năng xác định bit exception để loại trừ 1 trong 2 số ngõ vào là số vô cùng, và có chức năng xem bit zero được xác định 1 trong 2 số ngõ vào là số 0 hay không. Đồng thời thực hiện phép xor để xác định dấu sign của ngõ ra.
* EXPONENT\_MUL: có chức năng cộng 2 phần exponent lại với nhau. Và công thêm bit 47 của ngõ ra của bộ MULTIPLY.
* FLOW\_MUL: có chức năng kiểm tra overflow và underflow.
* RES\_MUL: có chức năng tổng hợp lại kết quả.

1. Nội dung của file MUL\_2021.v:

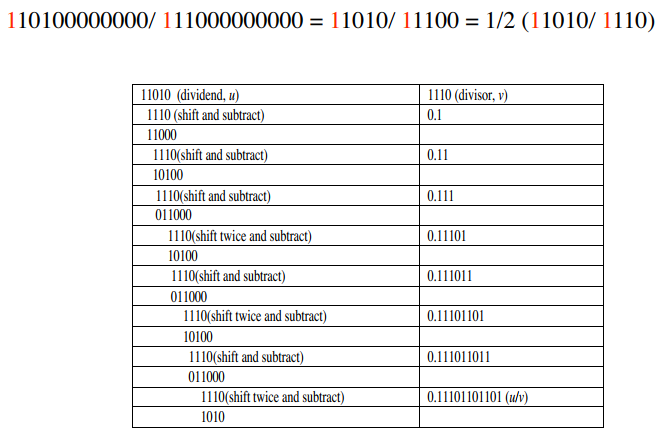


1. Nội dung khi testbench:



1. **Bộ chia:**
2. Giải thuật:

* Tách 2 số 32 bits của ngõ vào thành 3 thành phần bao gồm: sign, exponent và significand.
* Về cơ bản, việc thực hiện phép chia bản chất là thực hiện phép trừ phần exponent và phép chia cho phần significand và phép xor cho phần sign.
* Khi thực hiện phép chia cho phần significand. Về bản chất giống việc phép chia bình thường, hay nói cách khác là thực hiện phép dịch trái và phép trừ hang dọc xuống. Việc thực hiện phép chia 24 bits cũng tương tự như trường hợp dưới dây:

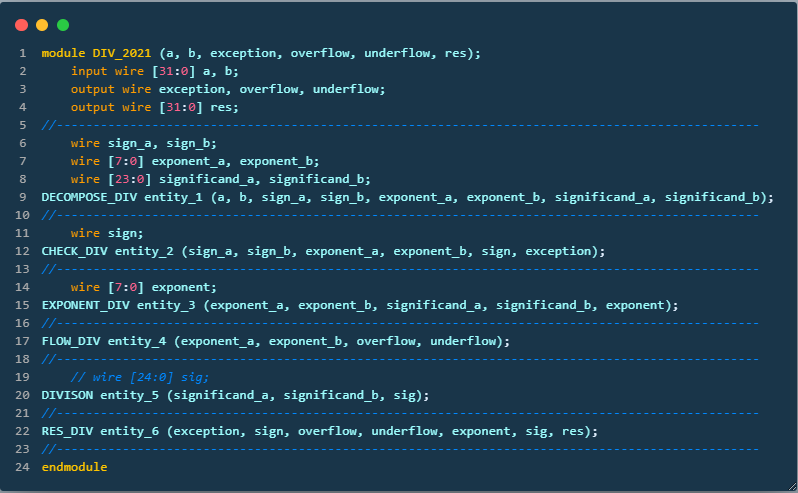


* Tổng hợp lại kết quả và kiểm tra xem kết quả ngõ ra có bị tràn trên hoặc tràn dưới hay không.

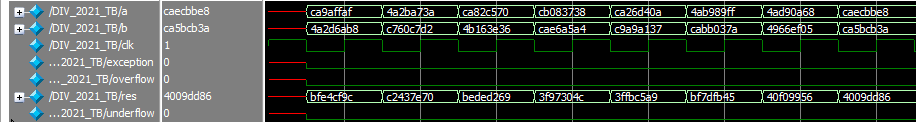
1. Các khối cơ bản: bao gồm 6 khối cơ bản như sau:

* DECOMPOSE\_DIV: có chức năng phân tích 2 số 32 bits ngõ vào thành 3 thành phần bao gồm: sign, exponent, significand. Trong đó phần significand được mở rộng thành 24 bits với bit thứ 23 là bit 1.
* CHECK\_DIV: có chức năng xác định bit exception để loại trừ 1 trong 2 số ngõ vào là số vô cùng. Đồng thời thực hiện phép xor để xác định dấu sign của ngõ ra.
* EXPONENT\_DIV: có chức năng trừ 2 phần exponent lại với nhau. Và trừ thêm 1bit nếu phần significand a nhỏ hơn phần significand b.
* FLOW\_DIV: có chức năng kiểm tra overflow và underflow.
* DIVISION: thực hiện phép chia phần significand. Với ngõ vào là 2 số 24 bits (phần significand dược mở rộng khi thêm bit thứ 23 là 1) và kết quả ngõ ra là 1 số 25 bits là kết quả của thương số.
* RES\_DIV: có chức năng tổng hợp lại kết quả.

1. Nội dung của file DIV\_2021.v:

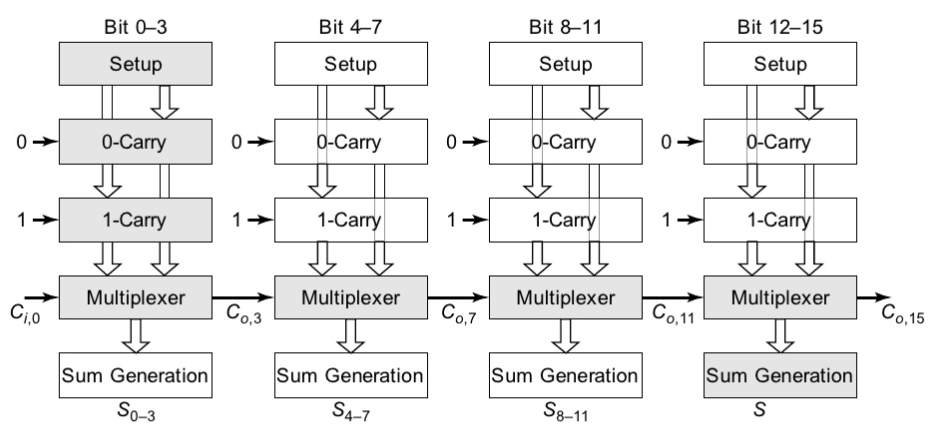


1. Nội dung khi testbench:



1. **Bộ cộng thứ hai:**
2. Giải thuật:

* Đặt vấn đề: Vấn đề của bộ cộng thứ nhất chính là các bộ full-adder phía sau phải chờ carry của bộ full-adder phía trước. Thời gian thực hiện bộ cộng thứ nhất cộng lại bằng thời gian n bộ full-adder.
* Bộ cộng thứ hai (carry-select-adder) bao gồm các bộ full-adder và các bộ multiplexer. Việc cộng 2 số n bits với carry-select-adder được thực hiện với hai bộ adder, 1 bộ adder với trường hợp bit carry bằng 1 và 1 bộ adder với trường hợp bit carry bằng 0. Sau đó 2 kết quả sẽ được cho qua 1 bộ multiplexer với carry khi cộng n bits trước đó để chọn ra kết quả đúng.
* Sơ đồ phần cứng bộ carry select adder 16 bits:

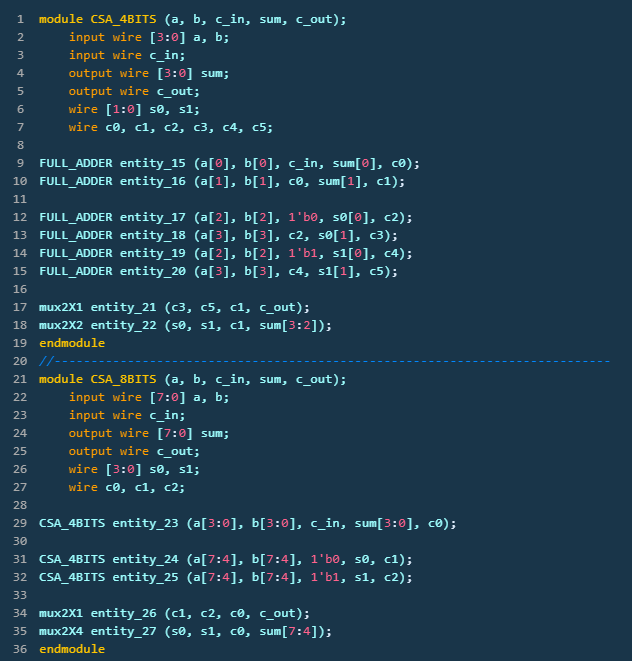


1. Các khối cơ bản:

* Ta sử dụng từ các khối cơ bản là FULL\_ADDER và mux2X1 đã được xây dựng ở các bộ quan trọng của phép toán.
* Xây dựng bộ CSA\_32BITS từ 2 bộ CSA\_16BITS, bộ CSA\_16BITS từ 2 bộ CSA\_8BITS, bộ CSA\_8BITS từ 2 bộ CSA\_4BITS.

1. Nội dung của file CSA\_ADD.v:

* Dưới đây là đoạn code verilog minh họa bộ dịch 4 bits, 8 bits. Các bộ CSA\_ADD lớn hơn như 16 bits, 32 bits đã được viết tương tự trong file code.

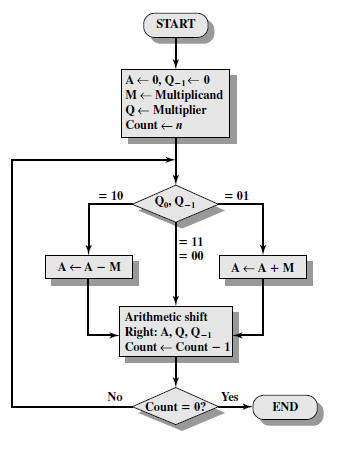


1. Nội dung khi testbench:



1. **Bộ nhân thứ hai:**
2. Giải thuật:

* Bộ nhân thứ hai được xây dựng với cấu trúc tuong tự như bộ nhân thứ nhất, chỉ thay đổi giải thuật ở phần nhân 2 significand vói nhau. Nhóm sử dụng thuật toán Booth để xử lý phần tính toán này.
* So vói thuật toán Shift-Add ở bộ nhân thứ nhất, thuật toán Booth sẽ tính nhanh hơn trong trường hợp nếu nhân tử có 1 dãy các bit 1 hoặc 1 dãy các bit 0 liên tiếp nhau.
* Ví dụ: khi sử dụng thuật toán Shift-Add nếu nhân tử có dạng 11111 thì ta phải nhân, dịch và cộng 5 lần. Tuy nhiên nếu trong thuật toán booth thì ta chỉ cần thực hiện phép nhân, dịch và cộng/trừ 2 lần (xét 10000-1).
* Dưới đây là lưu đồ giải thuật:

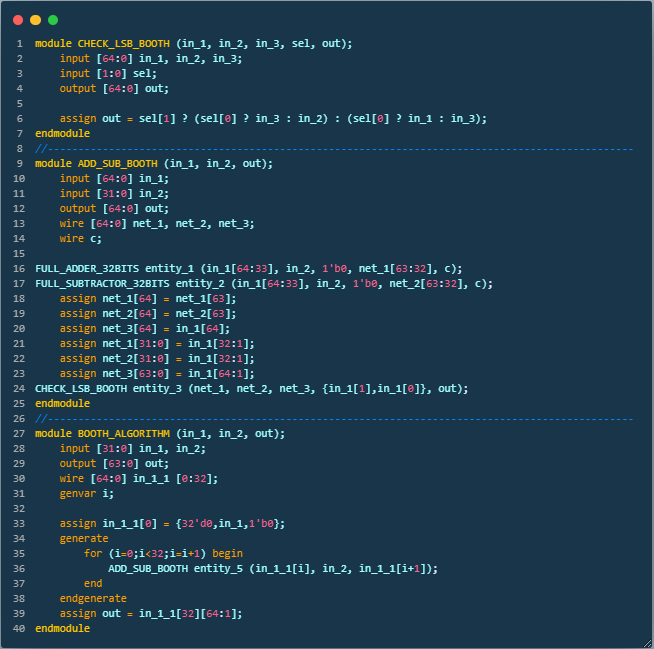


1. Các khối cơ bản:

* CHECK\_LSB\_BOOTH: khối này nhận 3 ngõ vào 65 bits và 1 ngõ vào 2 bits, ngõ ra của khối là 1 số 65 bits. Có chức năng kiểm tra 2 bits cuối cùng của ngõ vào. Từ đó quyết định ngõ ra cho phù hợp.
* ADD\_SUB\_BOOTH: Thực hiện phép cộng hoặc trừ hoặc giữ nguyên và dịch phải toàn bộ ngõ ra sau khi thực hiện phép tính.
* BOOTH\_ALGORITHM: khối chính của phép tính. Dùng để thực hiện vòng lặp for thực hiện khối ADD\_SUB\_BOOTH 32 lần. Ngõ ra là 1 số 64 bits.

1. Nội dung của file BOOTH\_ALGORITHM.v:

* Dưới đây là đoạn code verilog minh họa bộ dịch 64 bits. Các bộ BOOTH\_ALGORITHM lớn hơn như 16 bits, 32 bits đã được viết tương tự trong file code.

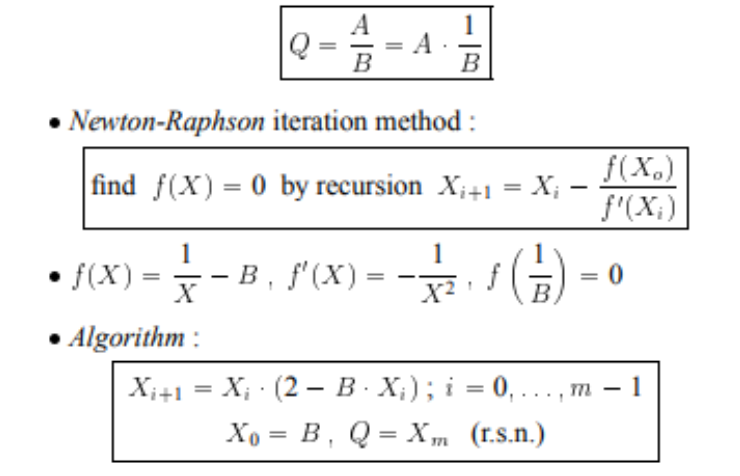


1. Nội dung khi testbench:



1. **Bộ chia thứ hai:**
2. Giải thuật:

* Bộ chia có ngõ vào là 2 số bị chia và số chia 32 bits, ngõ ra là thương 32 bits.
* Thay vì thực hiện phép A/B với A là số chia và B là số bị chia trực tiếp. Thuật toán ở đây đó là xấp xỉ 1/B bằng phương pháp Newton-Raphson rồi nhân với A bằng bộ nhân đã được xây dựng ở bộ nhân.

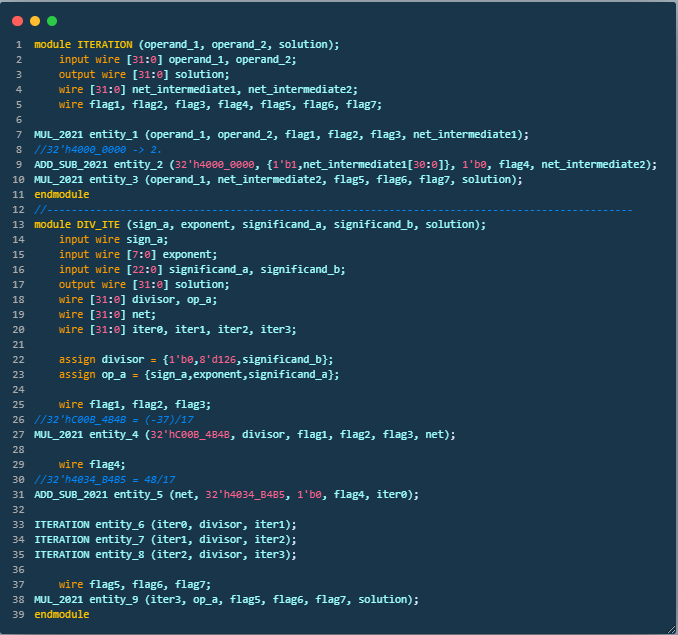


* Ta có hàm xấp xỉ tuyển tính: .

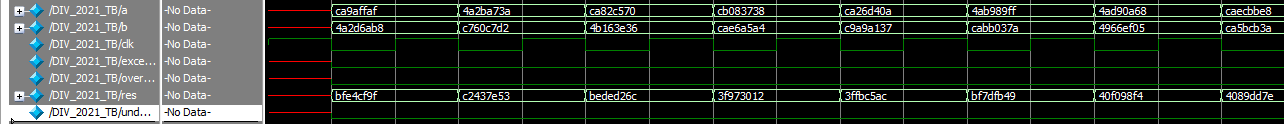
1. Các khối cơ bản:

* ITERATION: khối này có ngõ vào là 2 số 32 bits, ngõ ra là 1 số 32 bits. Có nhiệm vụ thực hiện việc xấp xỉ dung phương pháp Newton-Raphson.
* DIV\_ITE: khối chính của phép toán. Có nhiệm vụ thực hiện hàm xấp xỉ tuyển tính trên, sau đó thực hiện khối ITERATION rồi cuối cùng nhân với số chia A.

1. Nội dung của file DIV\_NEWTON\_RAPHSON.v:



1. Nội dung khi testbench:



1. **Bộ căn:**
2. Giải thuật:

* Bộ căn có ngõ vào là 1 số bị căn x và số bậc căn n, kết quả thu được là 1 số r sao cho . Các số x, r, n đều được biểu diễn dưới dạng số floating point 32 bits (1 bit sign, 8 bits exponent, 23 bits significand).
* Thuật toán nhóm sử dụng theo phương pháp khai triển Taylor-Maclaurin

và 1 số phép biến đổi đại số từ biểu thức .





* Để tính  ta sử dụng khai triển Taylor-Maclaurin:





* Công thức trên chỉ hội tụ tại . Nên để tính trường hợp  ta sẽ tính . Như vậy, ta sẽ xây dựng 1 bộ so sánh trị tuyệt đối số bị căn x với 1, nếu  ta sẽ tính , còn nếu  ta sẽ tính  rồi sau đó lấy bù
* Để tính giá trị , nhóm đã tính đến bậc 10 của chuỗi Taylor. Hải kết quả sẽ được qua 1 bộ mux2X32 để chọn 1 trong 2 số 32 bits dựa vào bit ngõ ra của bộ so sánh. Kết quả của bộ mux2X32 sẽ được đem chia với n để tính được .
* Từ đây ta tính được giá trị r: 
* Khai triển Taylor-Maclaurin của : 
* Khai triên trên hội tụ với mọi x nên không có trường hợp ngoại lệ. Nhóm đã tính giá trị của chuỗi Taylor trên đến bậc 10.
* Các trường hợp đặc biệt: , nhóm đã xây dựng các khối riêng lẻ so sánh x với các giá trị trên cho mỗi trường hợp.

1. Các khối cơ bản:

* COMPARE\_WITH\_1: có ngõ vào là 1 số 32 bits chính là số bị căn, ngõ ra là cờ báo 1 bit, làm nhiệm vụ xem trị tuyệt đối của số 32 bits đó có lớn hơn 1 hay không. Sử dụng khối COMPARATOR\_32BITS làm tiền đề.
* INVERSE: có ngõ vào là 1 số 32 bits chính là số bị căn, ngõ ra là 1 số 32 bits; có nhiệm vụ tính toán giá trị nghịch đảo của số hạng ngõ vào. Sử dụng khối DIV\_2021.
* NATURAL\_LOGARIT: có ngõ vào là 1 số 32 bits, có ngõ ra là 1 số 32 bits; làm nhiệm vụ tính toán giá trị logarit tự nhiên của 1 số 32 bits theo công thức Taylor-Maclaurin. Nhóm sử dụng các bộ ADD\_SUB\_2021, MUL\_2021, DIV\_2021.
* INVERT\_1: ngõ vào là 1 số 32 bits, ngõ ra là 1 số 32 bits; có nhiệm vụ lấy bù 1 của số ngõ vào.
* NATURAL\_EXPONENT: có ngõ vào là 1 số 32 bits, ngõ ra là 1 số 32 bits, có nhiệm vụ lấy lũy thừa e dựa theo chuỗi Taylor đã đề cập ở phần giải thuật.
* ERROR\_CHECKING: có ngõ vào là 1 số 32 bits, ngõ ra là 1 số 32 bits; có nhiệm vụ kiểm tra số bị căn có phải số âm hay không.
* CHECKING\_NAN: có ngõ vào là 3 số 32 bits gồm số bị căn, số bậc căn và kết quả của khối NATURAL\_EXPONENT; ngõ ra là 1 số 32 bits; có nhiệm vụ kiểm tra xem 2 số bị căn và số bậc căn có phải là NaN hay không thì ngõ ra sẽ là NaN.
* ZERO\_CHECKING: có ngõ vào là 3 số 32 bits gồm số bị căn, số bậc căn và kết quả của khối CHECKING\_NAN; ngõ ra là 1 số 32 bits; có nhiệm vụ kiểm tra xem 2 số bị căn và số bậc căn có phải là 0 hay không, nếu có thì ngõ ra sẽ bằng NaN nếu số bậc căn là 0, ngõ ra sẽ bằng 0 nếu số bậc căn khác 0 và số bị căn là 0, nếu cả 2 số trên khác 0 thì lấy kết quả từ khối trước.
* INFINITIY\_CHECKING: có ngõ vào là 3 số 32 bits gồm số bị căn, số bậc căn và kết quả của khối ZERO\_CHECKING; ngõ ra là 1 số 32 bits; có nhiệm vụ kiểm tra xem 2 số bị căn và số bậc căn có phải là số vô cùng hay không thì ngõ ra là số NaN, nếu cả 2 số đều khác 0 thì lấy kết quả từ khối trước.
* COMPOSE: có ngõ vào là 1 số 32 bits, 1 số bit dấu của số bị căn và ngõ ra là 1 số 32 bits, có nhiệm vụ bit dấu sign giống với bit dấu sign của số bị căn đã nhập ban đầu.