BÁO CÁO BÀI TẬP LỚN MÔN SỐ HỌC MÁY TÍNH

FLOATING POINT

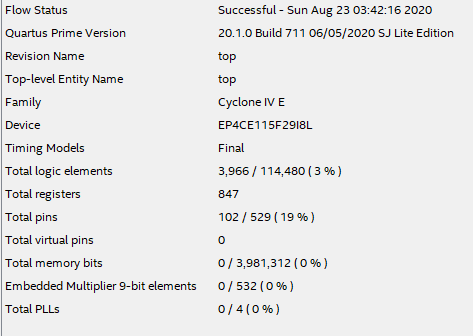
Nhóm 9

Trần Thị Thu Trang – 1870649

Trần Huỳnh Xuân Trường – 1970432

# Tổng hợp

Synthesis trên kit DE 2



Fmax ở 1000mV, 100C Model = 43.17 MHz

Fmax ở 1000mV, -40C Model = 45.14 MHz

* Khối cộng trừ cần 2 cycles, pipeline
* Khối nhân cần 3 cycles, pipeline
* Khối chia cần 13 cycles

# Adder - Subtractor module

Khối thực hiện phép toán trong 2 cycles, pipeine

## Phép cộng sử dụng CSA (Carry save adder)

Ở phép cộng 3 số thông thường a + b + c, ta sử dụng 1 bộ cộng cho cặp số (a+b) và 1 bộ cộng cho cặp số (a+b)+c, path delay trễ nhất ta phải mất là 2 lần delay của bộ cộng thông thường. Trong khi đó, phép cộng 3 số CSA giúp giảm delay của phép cộng thông thường bằng việc bằng việc tính trực tiếp chữ số của kết quả. Ví dụ

10111010101011011111000000001101 (a)

+ 11011110101011011011111011101111 (b)

+ 10010010101101110101001101010010 (c)

Sử dụng CSA:

10111010101011011111000000001101

+ 11011110101011011011111011101111

+ 00010010101101110101001101010010

= 21132130303123132223112112112222

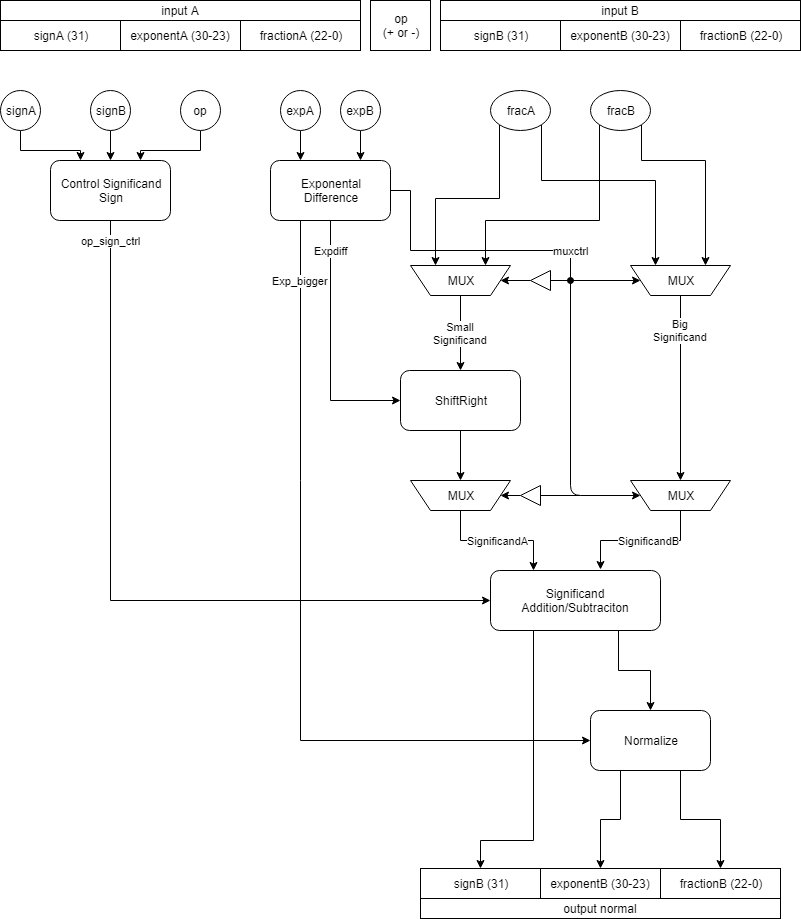
Bước hiệu chỉnh bằng việc cho qua khối cộng thông thường:

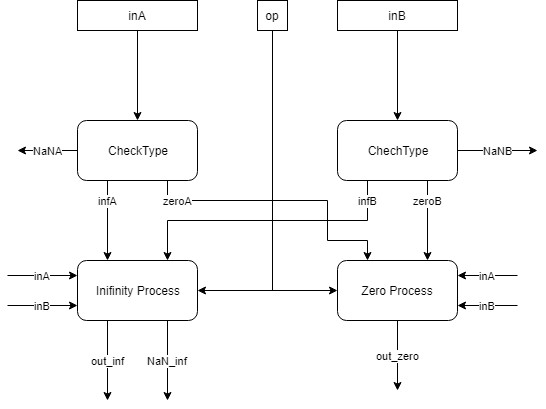
01110110101101110001110110110000 and

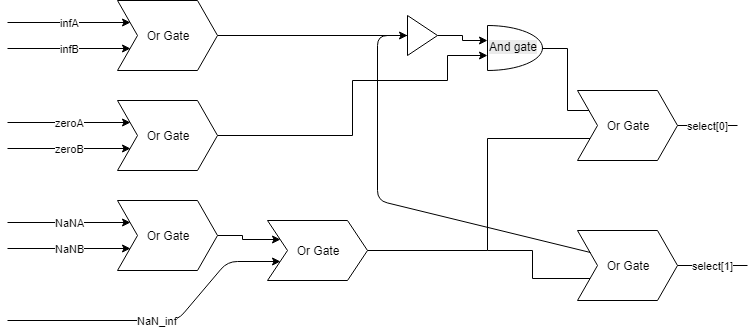
100110101010110111110010010011110

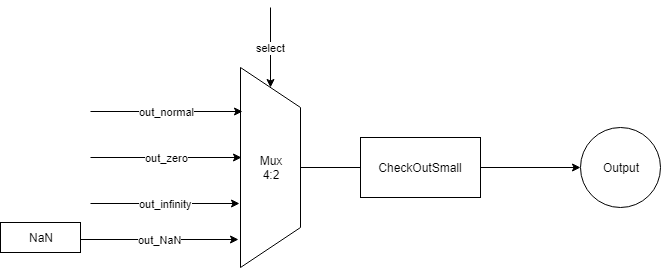
Theo như trên, CSA critical path chỉ phụ thuộc vào 1 delay của phép cộng thông thường và delay của khối Full Adder 🡺 nhỏ hơn phép cộng 3 số thông thường.

## Phép cộng trừ Floating Point









* Khối control significand sign:

Dựa vào dấu của các input và operation (+ hay -) để quyết định đảo bit các significand.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| signA | signB | Op | Opsign[1] | Opsign[0] |
| + | + | + | 0 | 0 |
| + | + | - | 1 | 0 |
| + | - | + | 1 | 0 |
| + | - | - | 0 | 0 |
| - | + | + | 0 | 1 |
| - | + | - | 1 | 1 |
| - | - | + | 1 | 1 |
| - | - | - | 0 | 1 |

Opsign[1] quyết định signficand B. Opsign[0] quyết định significand A. Opsign[1] hay [0] nếu bằng 1 thì đảo bit significand tương ứng.

* Khối Exponental Difference

Dùng tìm ra số mũ lớn hơn và chênh lệch về số mũ để dùng trong dịch significand và chuẩn hóa (normalize).

Trừ 2 số mũ để tìm chênh lệch, dùng dấu của kết quả trừ để biết số mũ nào lớn hơn.

Sau cùng là lấy bù 2 kết quả trừ nếu kết quả trừ trên bị âm.

* Chuẩn bị cho các significand

Significand của số có phần số mũ nhỏ hơn sẽ được lựa chọn thông qua các MUX để lái vào bộ dịch phải. Lượng bit dịch phải chính là chênh lệch về số mũ. Sau đó, các significand được trả về vị trí cũ cũng qua các MUX

* Khối Significand

Khối này dùng để cộng 2 phần significand lại với nhau.

Significand sẽ được đảo bit tùy thuộc vào opsign ở phần trước.

Các significand có 24 bit, kết quả trả về là 25 bit. Vì vậy trước khi cộng ta sign-extend các significand lên 26 bit, với bit thứ 26 dùng để xác định dấu ngõ ra.

Nếu bit dấu nói trên bằng 1(kết quả âm) , thực hiện lấy bù 2 để ra kết quả dương.

Các quá trình cộng sử dụng CSA.

* Khối normalize (chuẩn hóa)

Dùng để tìm số mũ và phần fraction phù hợp cho ngõ ra.

Các công đoạn:

* + Tìm bit 1 đầu tiên tính từ MSB đến LSB
  + Dựa vào vị trí bit 1 để lấy phần fraction và tính phần số mũ cần trừ xuống
  + Làm tròn phần fraction
  + Tìm số mũ ngõ ra.
  + Sau khối này ta có ngõ ra của bộ cộng trừ trong trường hợp ngõ vào là số bình thường( kah1c 0, inf hay NaN)”
  + Phần dấu có từ khối Significand
  + Phần mũ và fraction có từ khối Normalize.
* Xử lí với các trường hợp đặc biệt.

Các trường hợp đặc biệt:

* + Zero (0). Phần mũ và fraction bằng 0
  + Infinity: Phần mũ bằng 11111111, phần fraction bằng 0
  + NaN(not a number): Phần mũ bằng 11111111, phần fraction khác 0.

Khối CheckType có tác dụng kiểm tra ngõ vào A và B

Khối InifinityProcess dùng để tìm ngõ ra phù hợp trong trường hợp có ít nhất 1 ngõ vào là vô cùng:

* + Kết quả trả về là vô cùng nếu chỉ có 1 ngõ vào là vô cùng.
  + Nếu 2 ngõ vào đều là vô cùng, các phép tính toán cho kết quả inf – inf (vô định) sẽ được chuyển về NaN, ngược lại trả về vô cùng. Ngõ ra NaN\_inf là nhằm mục đích trên.
  + Khối ZeroProcess dùng để tìm ngõ ra phù hợp trong trường hợp có ít nhất 1 ngõ vào là 0:
  + Ngõ ra có thể là 0, inA hoặc inB và sẽ được lựa chọn thông qua bộ MUX.

Sau cùng: các ngõ ra gồm: out\_normal, out\_inf, out\_zero và outNaN (được gán thẳng bằng NaN) sẽ được đưa qua MUX 4-1, với tín hiệu lựa chọn là select.

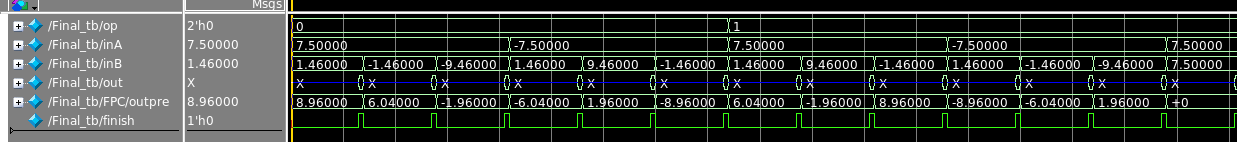
|  |  |  |  |
| --- | --- | --- | --- |
| Inf=infA+infB | Zero=zeroA+zeroB | NaN=NaNA+NaNB+NaN\_inf | Chọn |
| 0 | 0 | 0 | Out\_normal |
| 0 | 0 | 1 | Out\_NaN |
| 0 | 1 | 0 | Out\_zero |
| 0 | 1 | 1 | Out\_NaN |
| 1 | 0 | 0 | Out\_inf |
| 1 | 0 | 1 | Out\_NaN |
| 1 | 1 | 0 | Out\_inf |
| 1 | 1 | 1 | Out\_NaN |

Select[0] = select zero + select NaN = Zero./(inf) + NaN

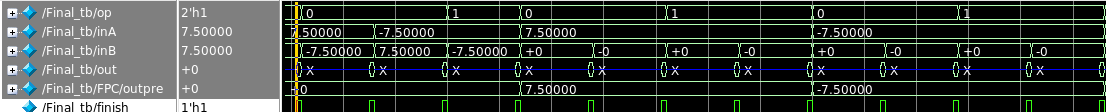
Select[1] = select inf + select NaN = NaN + inf

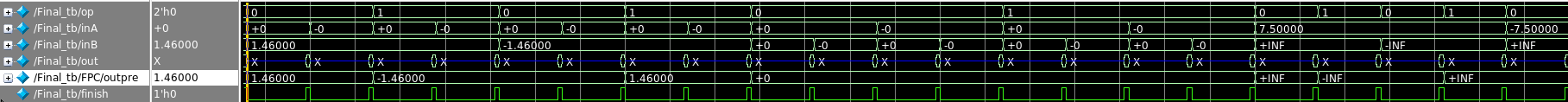
## Mô phỏng

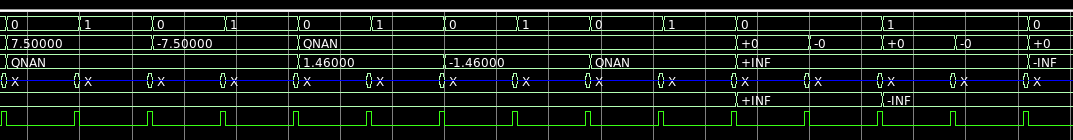
Tính thông thường:



Trường hợp đặc biệt:





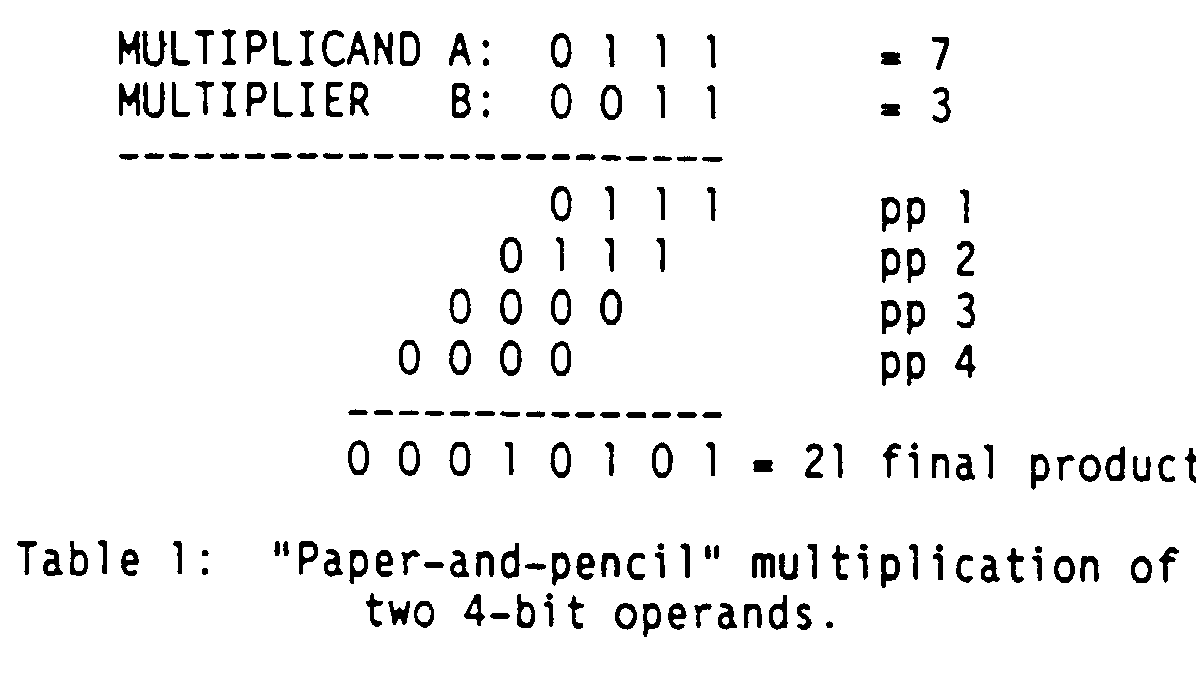


# Khối nhân

Khối thực hiện phép toán trong 3 cycles, pipeline

## Giải thuật Booth

Một phép nhân thông thường giữa hai số nhị phân có độ dài n bit được thực hiện trên cơ chế dịch và cộng. Ứng với phép nhân hai số n bit là n partition được tạo ra, sau đó chúng được cộng với nhau sau khi đã dịch bit một cách thích hợp.

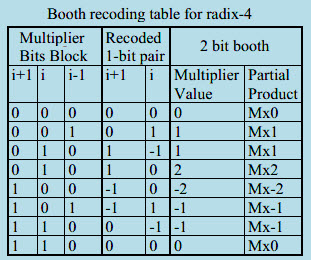


Số lượng partition càng nhiều thì lượng adder, hay lượng cổng logic phải sử dụng càng nhiều 🡺 Tìm kiếm một phương pháp nhân tạo ra ít partition hơn 🡺 Giải thuật Booth

Theo phép nhân thông thường, ứng với từng bit của multiplier có 1 partition. Giải thuật Booth thì khác, ứng với mỗi partition có thể sử dụng nhiều bit của multiplier. VD: radix-4 Booth sử dụng 3-bit cho 1 partition, radix-8 Booth sử dụng 4-bit cho một partition.

Partition được cấu thành từ càng nhiều bit của multiplier thì càng có thể mang nhiều giá trị khác nhau => đồng nghĩa với việc giải mã càng phức tạp hơn.

Trong khối tính Floating Point cho phép nhân, Radix-4 Booth được sử dụng với bảng giải mã như sau:



Cách thực hiện giải thuật radix-4 Booth:

* Mở rộng bit dấu để đảm bảo multiplier có số bit chẵn
* Chèn một bit 0 vào cuối của multiplier
* Lần lượt lấy từng nhóm 3 bit, kể từ bit 0 vừa chèn đến MSB của multiplier. Ứng với giá trị 3 bit này sẽ cho ra giá trị partition phù hợp. Chú ý rằng bit đầu tiên kể từ hướng từ LSB sang MSB của nhóm 3 bit phía sau là bit cưối cùng trong nhóm 3 bit phía trước đó.
* Sau khi các partition được tạo nên, chúng sẽ đước cộng lại với nhau theo quy tắc: partition kế tiếp được dịch trái 2 bit trước khi cộng với partition trước đó. Các partition khi cộng với nhau đều được extend bit dấu.

VD:

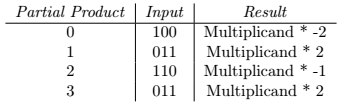


Multiplier là 01110110 cố số bit chẵn nên không cần extend bit dấu.

Chèn bit 0 vào cuối ta có: 011101100

4 nhóm 3 bit được gom lại theo thứ tự từ phải sang là: 100, 011, 110, 011

Dựa theo bảng giải mã, ta có:



Ta có giá trị các partition tương ứng như sau:

- 010100000

- 101100000

- 01010000

- 101100000

Kết quả phép cộng các partition:

Khi chuyển sang hệ thập phân ta có kết quả sau:



Giải thuật này được áp dụng cho phép nhân hai số floating point, cụ thể là phần nhân hai fraction.

Phần fraction có độ dài 23 bit, chén bit 1 vào phía trước để có significand chính xác. Sau đó chén thêm 2 bit 0 vào phía trước nhằm:

- Phần fraction được coi là các số không âm nên phải extend bit 0 để tránh nhầm lẫn

- Phép nhân 2 và -2 là tăng số lương bit lên 1 => chèn 2 bit 00

Như vậy, khối tính toán phần thập phân cần phép nhân 2 số dương 26 bit. Giải thuật Booth sẽ tạo nên tương ứng là 13 partition.

## Wallace Tree

Dùng giải thuật Booth đã giảm được lượng partition xuống còn 13 (so với 26 ban đầu). Tuy nhiên, để cộng được 13 partition trên cũng tốn khá nhiều tài nguyên, mà cụ thể ở đây là các full adder và half adder.

Wallace Tree là phương pháp tối giản hóa lượng adder đó. Thay vì nối các adder theo “input của bộ cộng sau cần kết quả của bộ cộng trước”, các bit của các partition được cộng theo kiểu đồng thời, miễn là chúng có trọng số ngang nhau.

Giải thuật cho Wallace Tree như sau:

- Cứ 3 bit có cùng trọng số (vị trí bit được xếp thẳng hàng) sẽ được input vào cho 1 full adder. Kết quả trả ra sẽ có 1 bit có trọng số không đổi (vị trí cũ) và 1 bit có trọng số cao hơn (vị trí bit bên trái)

- Nếu còn lại 2 bit cùng trọng số mà chưa được gom thì cho chúng vào 1 half adder, kết quả cũng ra 2 bit như trên

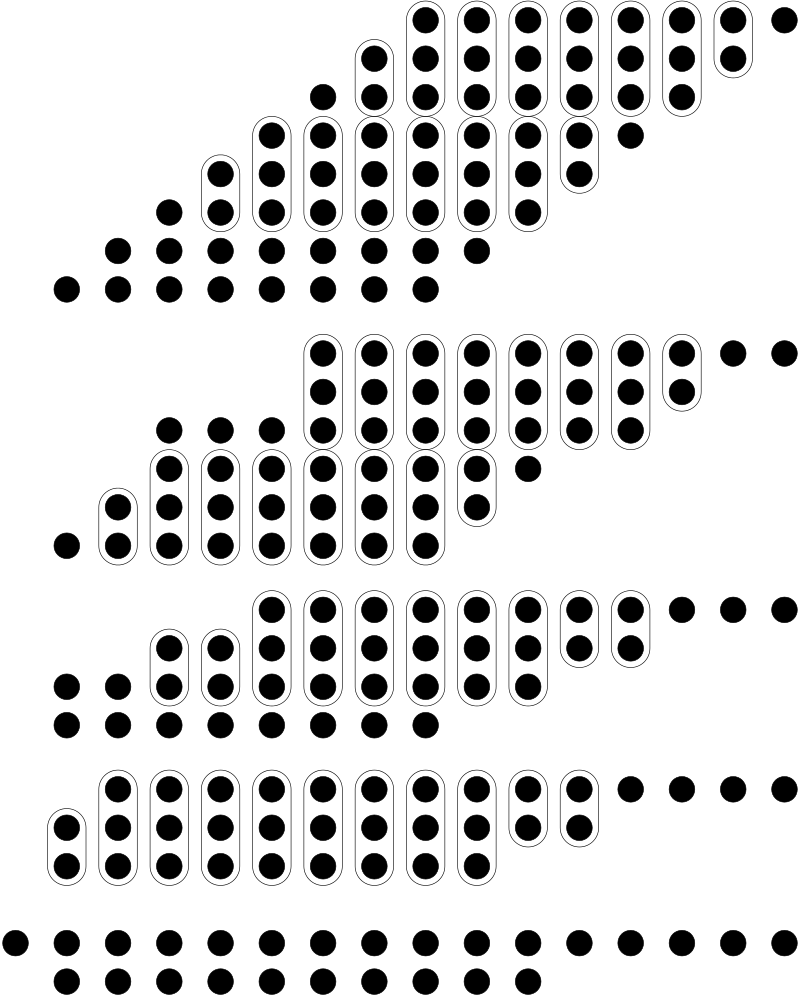
- Giữ nguyên trọng số nếu chỉ còn lại 1 bit.

- Thay thế các bit cũ đã được gom trước đó bằng các bit kết quả của các bộ adder. Ta có một phép cộng mới với số lượng partition đã giảm đi đáng kể

- Tiếp tục gom lại như ở bước đầu.

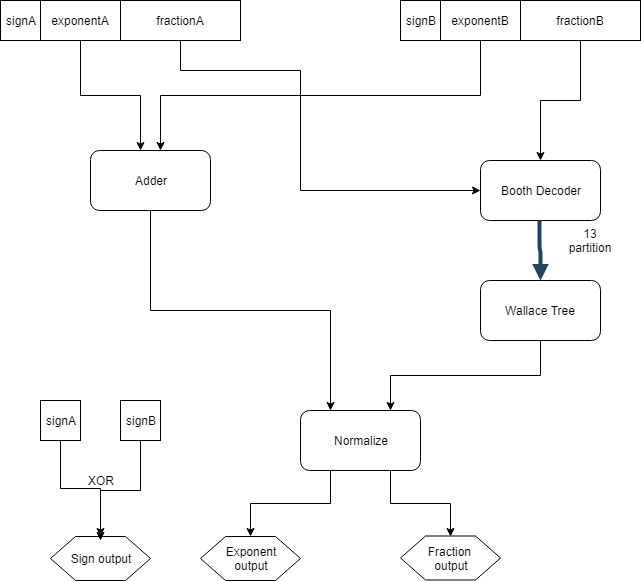
- Đến khi chỉ còn lại 2 partition thì cộng lại bình thường bằng các adder và xuất kết quả.

VD:

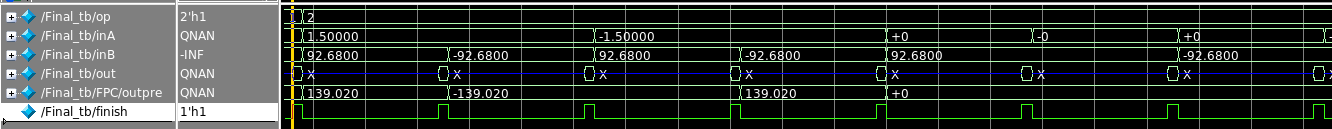


## Phép nhân Floating Point

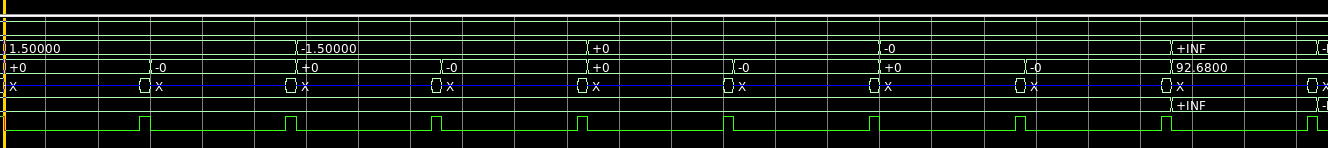
* Bộ nhân tổng quát:

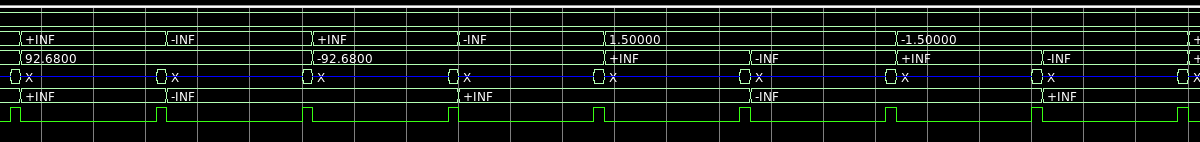


## Mô phỏng



Trường hợp đặt biệt:





# Khối chia

Khối thực hiện phép toán trong 13 cycles

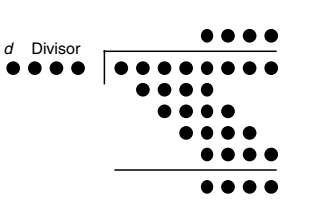
## Thuật toán chia high-radix:

Phép chia số z cho số d được thương q và số dư là s được biểu thị qua công thức:

Z = q\*d + s, trong đó sign(s) = sign(z) và |s| < |d|. (1)

Sơ đồ thực hiện phép chia:

q



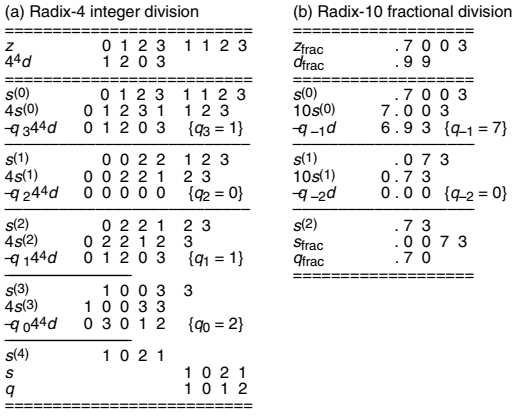
s5

z  
s1  
s2  
s3  
s4

Các chữ số của q được xác định theo:

với và (2)

Ví dụ:

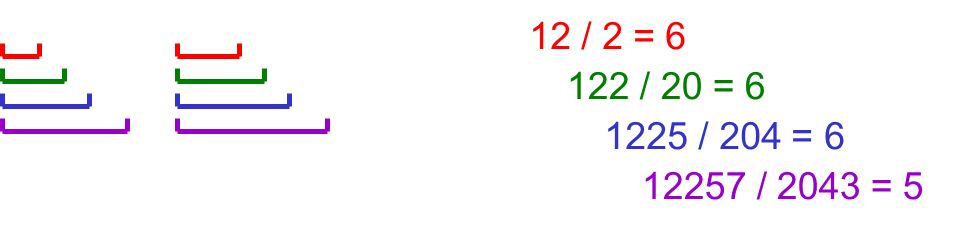


Radix thực tế thường có dạng r = 2b (do khi đó việc thực hiện phép nhân với r chỉ là dịch trái b bit nên dễ thiết kế hơn khi ). Nếu so sánh với radix-2 (hoặc thuật toán shift/subtract), thì với r = 2b sẽ tiết kiệm số chu kỳ tính toán hơn b lần.

Trong ví dụ (a) trên, nếu thực hiện bằng thuật toán shift/subtract thì sẽ mất 10 chu kỳ trong khi chỉ mất 5 chu kỳ với r=2b. Tuy nhiên việc tính ra chữ số tiếp theo trở nên phức tạp hơn (chọn 0,1,2 hay 3 trong khi với shift/subtract chỉ có 2 chữ số là 0 và 1). Một ví dụ khác để chỉ ra sự khó khăn khi chọn chữ số trong kết quả thương số, với r = 10:

**\_\_\_\_\_\_\_\_\_\_\_\_\_**

**2 0 4 3 | 1 2 2 5 7 9 6 8**



Trong ví dụ trên, sai số xảy ra khi ta chọn số các chữ số của z và d để tìm qk-j. Nhưng nếu tập biểu diễn của q trong ví dụ trên là {-9, -8, -7, … , 8, 9} ta có thể chọn thay vì thì sẽ tiết kiệm thời gian so sánh để tính toán chữ số của q (bởi vì ).

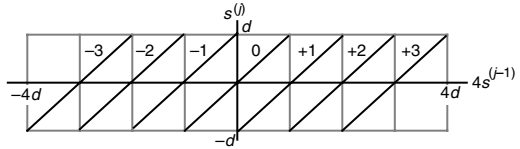
### Vùng overlap

Lấy ví dụ về r = 2. Biểu thức (2) trở thành:

với và (3)

Tập chữ số của q: {-3, -2, -1, 0, 1, 2, 3}

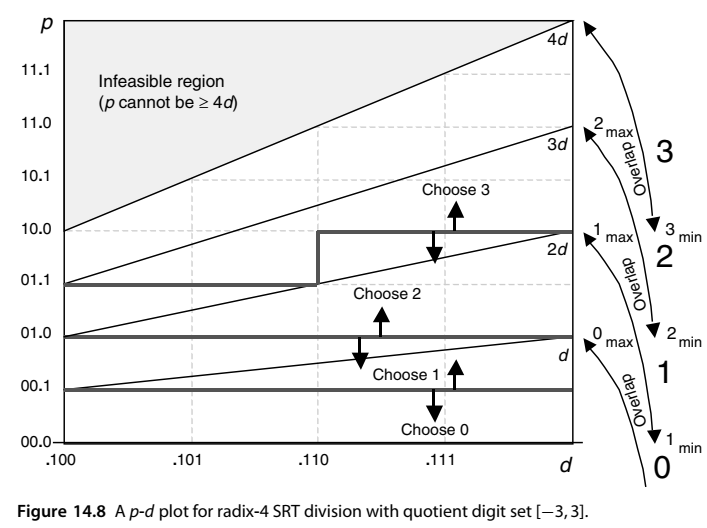
Từ đây ta có thể vẽ được đồ thị biểu diễn mối quan hệ giữa 4s(j-1) và s(j)



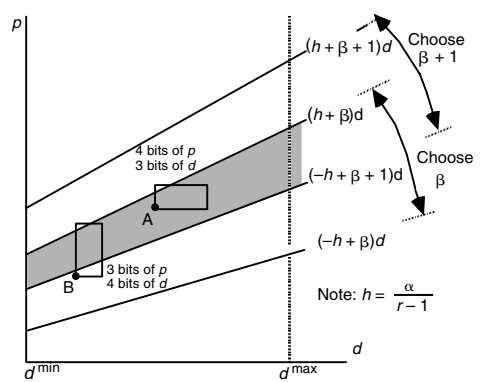
<---------><--------><--------><--------><--------><-------->

Ở đồ thị trên, ta thấy có 6 vùng mà ta có thể chọn chữ số kế tiếp là một trong 2 chữ số như vùng (-d ; 0) ta có thể chọn -1 hoặc 0, vùng (2d; 3d) ta có thể chọn 2 hoặc 3. Việc chọn chữ số khác nhau sẽ dẫn đến biểu diễn kết qủa q khác nhau nhưng các cách biểu diễn có có cùng giá trị.

Một đồ thị khác biểu diễn các vùng overlap và có thể dung nó để làm công cụ xác định chữ số của q, chính là đồ thị p-d (biểu diễn rs(j-1) với d) (đồ thị sau chỉ vẽ với 2s(j-1) dương vì phần âm có thể làm tương tự).



* Đồ thị p-d có thể được vẽ với trục ngang từ giá trị (dmin; dmax) tùy ý theo phạm vi bài toán
* Đường phân định chọn chữ số như theo đồ thị trên được xác định theo quy tắc:
  + Ta chọn số bit MSB biểu diễn cho p và d để biểu thị mức của trục tung, trục hoành. Giả sử ta chọn là 4 và 3.
  + Một số A được vẽ lên đồ thị p-d cùng với những số có cùng các số MSB tập hợp thành hình chữ nhật “điểm A” như hình sau.



* + Ta nói việc chọn số bit trên là khả thi khi hình chữ nhật của số A không cắt quá 1 đường và (h+ và h + -1 là 2 chữ số liên tiếp trong tập biểu diễn q có vùng overlap). Trong ví dụ trên việc chọn 3 bits cho biểu diễn p và 4 bits cho biểu diễn d là không khả thi khi hình chữ nhật “điểm B” cắt quá 1 đường ranh giới vùng overlap.

### Cách vẽ đồ thị p-d để xác định chữ số của q

Việc tìm đồ thị p-d để xác định tùy thuộc vào tập biểu diễn các chữ số của q. Một cách tổng quát với radix r=2b ta có tập biểu diễn là {-a, -a+1, … , 0, … , a-1, a}. Với a thỏa:

(4)

(5)

Theo đó, r = 4 ngoài tập {-3, -2, -1, 0, 1 2, 3}, q còn có thể biểu diễn với tập {-2; -1, 0, 1, 2}.

Mối quan hệ giữa rs(j-1) (partial remainder, p) và d là:

(6)

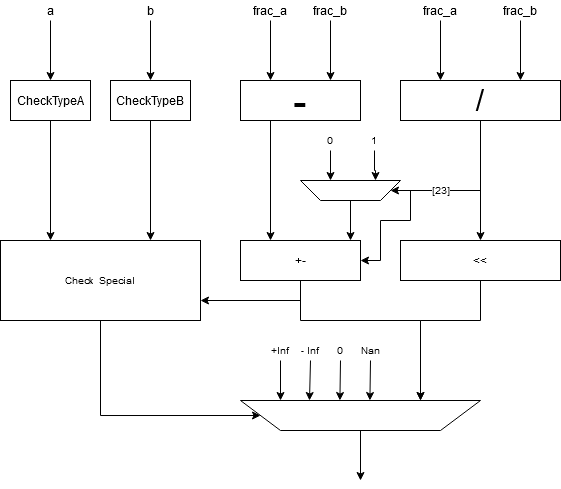
r = 4, từ (6) và phần trước ta có được đồ thị:

Các đường phân cách vùng overlap là (h+b)d với b {-3, -2,… 2, 3}.

Ta xác định đường phân cách chọn chữ số q và đường này phải nằm giữa vùng overlap của 2 chữ số.

## Datapath cho khối chia trong khối tính FP

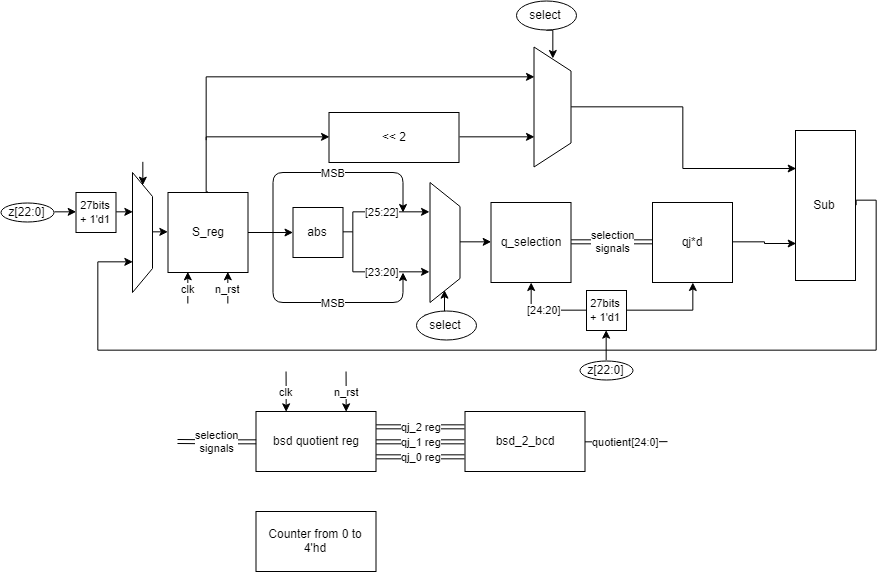
Datapath cho khối tính chia



Khối CheckType kiểm tra các input đầu vào có phải dạng đặt biệt hay không

Khối CheckSpecial cùng với khối CheckType có nhiệm vụ xác định các trường hợp đặc biệt khi thực hiện chia 2 số FP (Phép chia A/B)

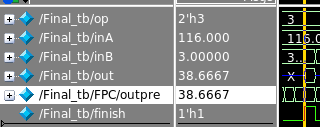
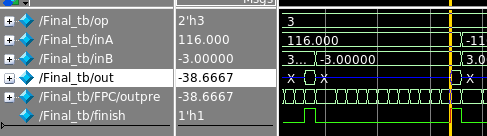
|  |  |  |
| --- | --- | --- |
| A | B | Output |
| 0 | X > 0 | 0 |
| X <> 0 | 0 | Inf |
| 0 | 0 | NaN |
| Inf | X | Inf \* (sign(A) ^ sign(B)) |
| X | Inf | 0 |
| Inf | Inf | NaN |
| A | B | Inf, 0 (nếu số mũ exp(A) +- exp(B) (-1) thỏa) |

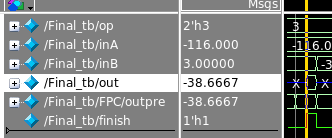


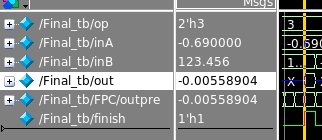
Ở cycle thứ nhất

## Mô phỏng

Tính thông thường





Trường hợp đặt biệt

