ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

KIẾN TRÚC MÁY TÍNH (CO2007)

Bài tập lớn

Đề tài 2

Giảng viên hướng dẫn: Trần Thanh Bình
Sinh viên: Bùi Nguyễn Thành Luân - 2111700
Đỗ Nguyễn An Huy - 2110193

THÀNH PHỐ HỒ CHÍ MINH, 2022

Mục lục

1 Danh sách thành viên & Phân công công việc

STT	Họ và tên	MSSV	Công việc
1	Bùi Nguyễn Thành Luân	2111700	Phép nhân
2	Đỗ Nguyễn An Huy	2110193	Phép chia

2 Yêu cầu

Đề số 2. Cho 2 số thực dạng chuẩn (Standard Floating Point) A và B với độ chính xác kép (64 bit). Sử dụng hợp ngữ assembly MIPS, viết thủ tục nhân, chia hai số A, B. Hiển thị input và output của phép tính.

3 Giải pháp hiện thực

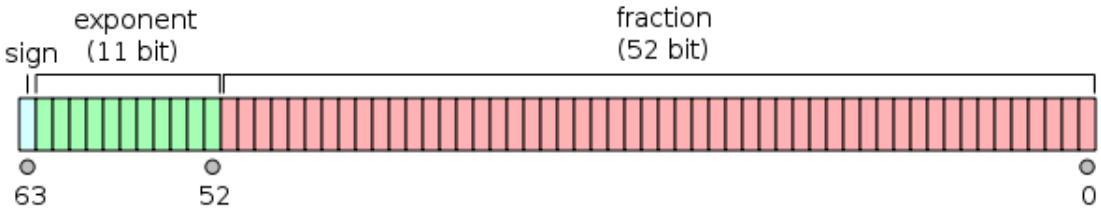
Theo chuẩn IEEE754, số thực với độ chính xác kép chiếm 64 bit và gồm 3 trường:

- Trường dấu chiếm 1 bit đầu tiên. (ký hiệu giá trị của bit này là s)
- Trường mũ chiếm 11 bit tiếp theo. (ký hiệu số nguyên không dấu được biểu diễn bởi 11 bit này là e)
- Trường thập phân chiếm 52 bit cuối cùng. (ký hiệu chuỗi 52 bit này là fraction)

Khi đó, nếu $1 \leq e \leq 2^{11} - 2$ thì số thực được biểu diễn bởi 64 bit này là

$$\mathbf{f} = (-1)^{\mathbf{s}} \times (1 + \overline{0.\mathbf{fraction}}) \times 2^{\mathbf{e}-\mathbf{bias}}$$

Trong đó, $\mathbf{bias} = 2^{10} - 1$.



Hình 1: Biểu diễn của số thực với độ chính xác kép theo chuẩn IEEE754

Với kiến trúc MIPS trong chương trình học, mỗi thanh ghi trong bộ xử lý chỉ có 32 bit, vì vậy cần 2 thanh ghi để lưu được 1 chuỗi bit biểu diễn số thực với độ chính xác kép. ***Ta sẽ xây dựng một số hàm để có thể trích xuất 3 trường này từ những bit được lưu trữ trong 2 thanh ghi.***

Ta giả sử đã trích xuất được các trường của số thực chính xác kép thứ nhất $\mathbf{f}_1$, lần lượt là bit dấu $\mathbf{s}_1$, số nguyên không dấu được biểu diễn bởi trường mũ (chưa trừ **bias**) $\mathbf{e}_1$ và chuỗi bit

phần thập phân $\mathbf{fraction}_1$. Tương tự với số thực chính xác kép thứ hai $\mathbf{f}_2$, ta trích xuất được $\mathbf{s}_2$, $\mathbf{e}_2$ và chuỗi bit phần thập phân $\mathbf{fraction}_2$.

Kết quả các phép toán trong hai hàm hiện thực sẽ được làm tròn theo hướng về gần 0.

3.1 Phép nhân

Phép toán ta cần thực hiện:

$$\begin{aligned}\mathbf{f}_1 \times \mathbf{f}_2 &= (-1)^{\mathbf{s}_1} \times (1 + \overline{0.\mathbf{fraction}_1}) \times 2^{\mathbf{e}_1 - \mathbf{bias}} \times (-1)^{\mathbf{s}_2} \times (1 + \overline{0.\mathbf{fraction}_2}) \times 2^{\mathbf{e}_2 - \mathbf{bias}} \\ &= (-1)^{\mathbf{s}_1 + \mathbf{s}_2} \times \overline{1.\mathbf{fraction}_1} \times \overline{1.\mathbf{fraction}_2} \times 2^{\mathbf{e}_1 + \mathbf{e}_2 - 2 \times \mathbf{bias}} \\ &= (-1)^{\mathbf{s}_1 + \mathbf{s}_2} \times (\overline{1\mathbf{fraction}_1} \times \overline{1\mathbf{fraction}_2} \times 2^{-104}) \times 2^{\mathbf{e}_1 + \mathbf{e}_2 - 2 \times \mathbf{bias}}\end{aligned}$$

Kết quả của phép toán được tính toán như sau:

1. Tính toán số mũ ban đầu của phép toán $\mathbf{e} = \mathbf{e}_1 + \mathbf{e}_2 - \mathbf{bias}$.
2. Thực hiện phép nhân 2 số tự nhiên trong hệ nhị phân $i_1 = \overline{1\mathbf{fraction}_1}$ và $i_2 = \overline{1\mathbf{fraction}_2}$ và chia kết quả cho 2^{104}, ta thu được một số thực $\mathbf{r}$.

   Dễ thấy $1_2 \leq \mathbf{r} < 100_2$, vì i_1 và i_2 có 53 bit, phép nhân giữa i_1 và i_2 sẽ có không quá 106 bit, sau đó ta lại chia kết quả cho 2^{104}.
3. Làm tròn và chuẩn hóa.

   Bởi vì ở đây, ta làm tròn kết quả về gần 0, ta chỉ đơn giản là bỏ đi các bit mà phần cứng không thể biểu diễn được.

   Theo nhận xét ở trên, $1_2 \leq \mathbf{r} < 100_2$ nên trong trường hợp phải thực hiện chuẩn hóa, tệ nhất chỉ cần dịch phải r 1 bit và cộng số mũ ban đầu thêm 1 đơn vị.

   Sau khi thực hiện làm tròn và chuẩn hóa, ta thu được $\mathbf{r}_{norm}$ và $\mathbf{e}_{norm}$.
4. Kiểm tra tràn số.

   Nếu $\mathbf{e}_{norm} \geq 2^{11} - 1$, kết quả là vô cực (dấu được xác định sau).

   Nếu $\mathbf{e}_{norm} \leq 0$, kết quả là 0 (dấu được xác định sau).
5. Xác định bit dấu của phép toán.

   $\mathbf{s} = 0$ nếu $\mathbf{e}_1 = \mathbf{e}_2$ và 1 nếu ngược lại.
6. Kết quả xấp xỉ của phép toán là một chuỗi 64 bit trong đó:
   - Trường dấu bằng $\mathbf{s}$.
   - Trường mũ bằng $\mathbf{e}_{norm}$.
   - Trường thập phân bằng $\mathbf{r}_{norm} - 1$.

3.2 Phép chia

Phép toán ta cần thực hiện:

$$\begin{aligned}\frac{\mathbf{f}_1}{\mathbf{f}_2} &= \frac{(-1)^{\mathbf{s}_1} \times (1 + \overline{0.\mathbf{fraction}_1}) \times 2^{\mathbf{e}_1 - \mathbf{bias}}}{(-1)^{\mathbf{s}_2} \times (1 + \overline{0.\mathbf{fraction}_2}) \times 2^{\mathbf{e}_2 - \mathbf{bias}}} \\ &= (-1)^{\mathbf{s}_1 - \mathbf{s}_2} \times \frac{\overline{1.\mathbf{fraction}_1}}{\overline{1.\mathbf{fraction}_2}} \times 2^{\mathbf{e}_1 - \mathbf{e}_2} \\ &= (-1)^{\mathbf{s}_1 - \mathbf{s}_2} \times \frac{\overline{1\mathbf{fraction}_1}}{\overline{1\mathbf{fraction}_2}} \times 2^{\mathbf{e}_1 - \mathbf{e}_2}\end{aligned}$$

Kết quả của phép toán được tính toán như sau:

1. Tính toán số mũ ban đầu của phép toán $\mathbf{e} = \mathbf{e}_1 - \mathbf{e}_2 + \mathbf{bias}$.
2. Thực hiện phép chia 2 số tự nhiên trong hệ nhị phân $i_1 = \overline{1\mathbf{fraction}_1}$ cho $i_2 = \overline{1\mathbf{fraction}_2}$, ta thu được một số thực r.

   Dễ thấy $0.1_2 \leq \mathbf{r} < 10_2$, vì i_1 và i_2 có số chữ số bằng nhau (53 bit, tính thêm bit 1 đầu tiên).
3. Làm tròn và chuẩn hóa.

   Lưu ý rằng, phần cứng chỉ có thể biểu diễn được hữu hạn các bit. Bởi vì ở đây, ta làm tròn kết quả về gần 0, sau khi được làm tròn $0.1_2 < \mathbf{r}_{round} < 10_2$.

   Như vậy, trong trường hợp phải thực hiện chuẩn hóa, tệ nhất ta chỉ cần dịch trái $\mathbf{r}_{round}$ một bit và trừ đi ở số mũ ban đầu vừa tính toán được 1 đơn vị.

   Sau khi thực hiện làm tròn và chuẩn hóa, ta thu được $\mathbf{r}_{norm}$ và $\mathbf{e}_{norm}$.
4. Kiểm tra tràn số.

   Nếu $\mathbf{e}_{norm} \geq 2^{11} - 1$, kết quả là vô cực (dấu được xác định sau).

   Nếu $\mathbf{e}_{norm} \leq 0$, kết quả là 0 (dấu được xác định sau).
5. Xác định bit dấu của phép toán.

   $\mathbf{s} = 0$ nếu $\mathbf{e}_1 = \mathbf{e}_2$ và 1 nếu ngược lại.
6. Kết quả xấp xỉ của phép toán là một chuỗi 64 bit trong đó:
   - Trường dấu bằng $\mathbf{s}$.
   - Trường mũ bằng $\mathbf{e}_{norm}$.
   - Trường thập phân bằng $\mathbf{r}_{norm} - 1$.

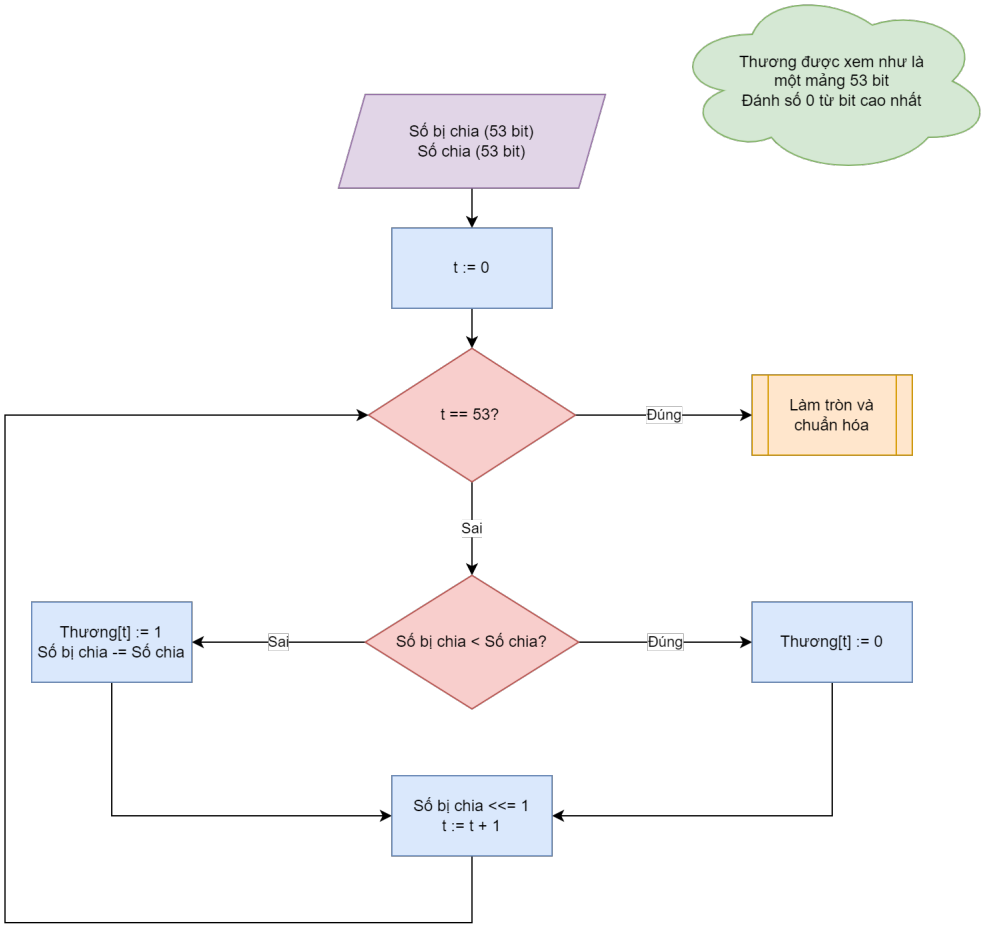
Các trường hợp đặc biệt:

- Số bị chia và số chia đều bằng 0, ta trả về NaN.
- Số bị chia bằng 0, số chia khác 0, ta trả về 0, dấu được xác định bởi trường dấu của 2 số.
- Số bị chia khác 0 và số chia bằng 0, ta trả về Infinity, dấu được xác định bởi trường dấu của 2 số.
- Ngoài ra, số bị chia và số chia có thể là NaN, Infinity và denorms (trừ số 0). ***Hàm thực hiện phép chia ở đây sẽ giả thiết rằng các trường hợp này không xảy ra***.

Tiếp theo, nhóm sẽ trình bày cụ thể 2 bước đó là: chia 2 số tự nhiên và làm tròn, chuẩn hóa.

3.2.1 Thao tác chia 2 số tự nhiên

Ở đây, nhóm trình bày ý tưởng thực hiện thao tác chia 2 số tự nhiên trong hệ nhị phân (cụ thể là 2 số nguyên 53 bit).
Giải thuật chia ở đây tương tự như phép đặt tính rồi tính thông thường.
Lưu ý rằng, chỉ có bit lớn nhất của thương trong sơ đồ khối trên là nằm trước dấu chấm, từ bit thứ 2 (đánh số 1) trở đi đều là những bit nằm sau dấu chấm.



Hình 2: Lưu đồ mô tả thao tác chia 2 số tự nhiên 53 bit trong hệ nhị phân

3.2.2 Thao tác làm tròn và chuẩn hóa

Nếu bit đầu tiên của thương khác 0 thì bit tiếp của thương phải khác 0. Ta chỉ cần dịch trái thương một bit và trừ đi số mũ ban đầu 1 đơn vị. Bởi vì các bit bị dịch sang trái, ta sẽ thực hiện thêm 1 vòng lặp ở bước chia 2 số nguyên để cập nhật bit thứ 53 (được đánh số 52).
Ở đây, thương được làm tròn về gần 0 nên ta chỉ cần dừng phép chia khi thương đã đủ 53 bit.

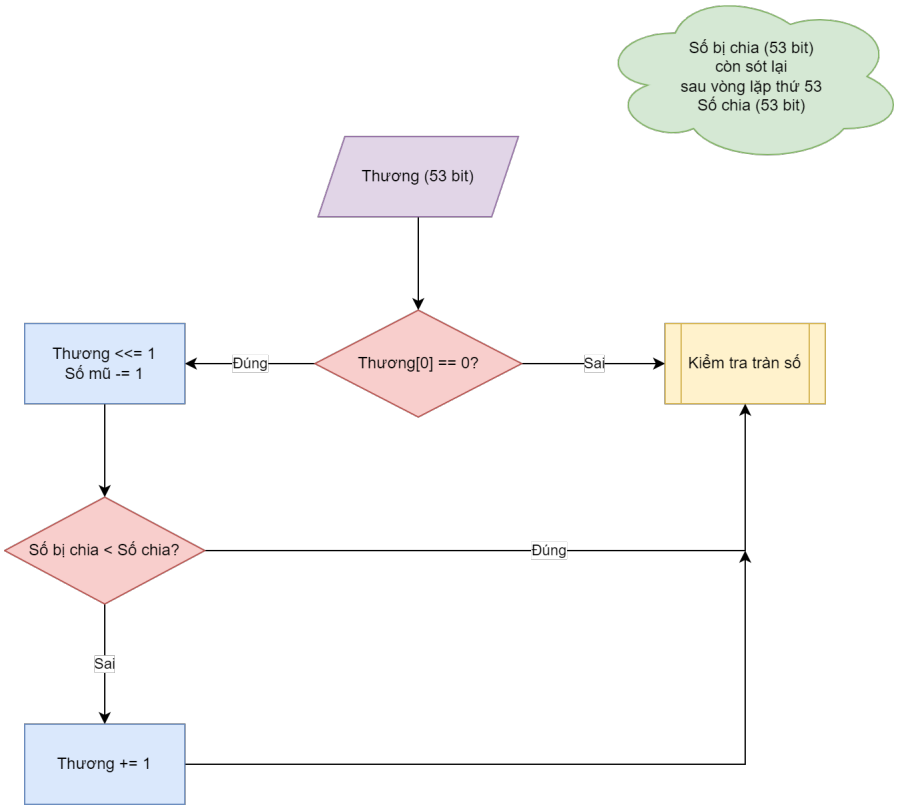
4 Hiện thực

4.1 Một số thao tác chung

4.1.1 Đọc số thực với độ chính xác kép

Tên hàm: readDouble

- Nhiệm vụ: Thực hiện đọc một số thực chính xác kép từ bàn phím.



Hình 3: Lưu đồ mô tả thao tác làm tròn và chuẩn hóa thương

- Input: Không có
- Ouput:
  - Thanh ghi $\$v_1$ chứa 32 bit cao của số thực.
  - Thanh ghi $\$v_0$ chứa 32 bit thấp của số thực.

```
### Read a double from keyboard
### Input  : None
### Output : $v1 -> upper 32 bits of the double
###          $v0 -> lower 32 bits of the double
readDouble:
    addi $v0, $v0, 7
    syscall
    mfc1.d $v0, $f0
    jr $ra
```

4.1.2 Xuất số thực với độ chính xác kép

Tên hàm: writeDouble

- Nhiệm vụ: Thực hiện xuất một số thực chính xác kép lên màn hình.
- Input:
  - Thanh ghi a_1 chứa 32 bit cao của số thực.
  - Thanh ghi a_0 chứa 32 bit thấp của số thực.
- Output: Không có

```
### Print a double to screen
### Input  : $a1 -> upper 32 bits of the double
###          $a0 -> lower 32 bits of the double
### Output : None
writeDouble:
    mtc1.d $a0, $f12
    addi $v0, $v0, 3
    syscall
    jr $ra
```

4.1.3 Trích bit dấu của số thực độ chính xác kép

Tên hàm: extractSign

- Nhiệm vụ: Trích bit dấu của số thực độ chính xác kép.
- Input:
  - Thanh ghi a_1 chứa 32 bit cao của số thực.
  - Thanh ghi a_0 chứa 32 bit thấp của số thực.
- Output: Thanh ghi v_0 bằng 1 nếu số thực âm, bằng 0 nếu ngược lại.
- Hàm đảm bảo không thay đổi giá trị của thanh ghi a_0 và a_1.

```
### Extract the sign bit of the double
### Input  : $a1 -> upper 32 bits of the double
###          $a0 -> lower 32 bits of the double
### Output : $v0 -> 1 if the double is negative, 0 otherwise
### Guarantees : Does not alter $a0 and $a1
extractSign:
    srl $v0, $a1, 31
    jr $ra
```

4.1.4 Trích trường mũ (chưa trừ bias) của số thực độ chính xác kép

Tên hàm: extractBiasedExponent

- Nhiệm vụ: Trích trường mũ (chưa trừ bias) của số thực độ chính xác kép.
- Input:

  - Thanh ghi a_1 chứa 32 bit cao của số thực.
  - Thanh ghi a_0 chứa 32 bit thấp của số thực.
- Output: Thanh ghi v_0 chứa giá trị của trường mũ (chưa trừ bias).
- Hàm đảm bảo không thay đổi giá trị của thanh ghi a_0 và a_1.

```
### Extract the biased exponent field of the double
### Input  : $a1 -> upper 32 bits of the double
###          $a0 -> lower 32 bits of the double
### Output : $v0 -> the biased exponent field of the double
### Guarantees : Does not alter $a0 and $a1
extractBiasedExponent:
   lui $t0, 0x7FF0      # bit mask 0x7FF0 0000
   and $v0, $a1, $t0
   srl $v0, $v0, 20
   jr $ra
```

4.1.5 Trích trường fraction của số thực độ chính xác kép

Tên hàm: extractFraction

- Nhiệm vụ: Trích trường fraction của số thực độ chính xác kép.
- Input:
  - Thanh ghi a_1 chứa 32 bit cao của số thực.
  - Thanh ghi a_0 chứa 32 bit thấp của số thực.
- Output:
  - Thanh ghi v_1 chứa 20 bit cao của phần thập phân và bit thứ 21 có giá trị bằng 1, những bit còn lại bằng 0.
  - Thanh ghi v_0 chứa 32 bit thấp của phần thập phân.
- Hàm đảm bảo không thay đổi giá trị của thanh ghi a_0 và a_1.

```
### Extract the fraction field of the double
### Input  : $a1 -> upper 32 bits of the double
###          $a0 -> lower 32 bits of the double
### Output : $v1 -> the upper 20 bits of the fraction
###          $v0 -> the lower 32 bits of the fraction
### Guarantees : Does not alter $a0 and $a1
extractFraction:
   lui $t0, 0x000F
   ori $t0, $t0, 0xFFFF     # bit mask 0x000F FFFF
   and $v1, $a1, $t0
   or $v0, $a0, $zero
   jr $ra
```

4.1.6 Dịch trái chuỗi 64 bit lưu trong 2 thanh ghi

Tên hàm: shiftLeft2Registers

- Nhiệm vụ: Dịch trái chuỗi 64 bit lưu trong 2 thanh ghi.
- Input:
  - Thanh ghi $\$a_0$ chứa nửa trái 32 bit của chuỗi.
  - Thanh ghi $\$a_1$ chứa nửa phải 32 bit của chuỗi.
- Output:
  - Thanh ghi $\$v_0$ chứa nửa trái 32 bit của chuỗi sau khi dịch trái 1 đơn vị.
  - Thanh ghi $\$v_1$ chứa nửa phải 32 bit của chuỗi sau khi dịch trái 1 đơn vị.
- Hàm đảm bảo không thay đổi bất kì giá trị của thanh ghi nào trừ thanh ghi $\$v_0$ và $\$v_1$.

```
### Shift left logical the bits across 2 registers
### Input  : $a0 -> the "left" register
###          $a1 -> the "right" register
### Output : $v0 -> the "left" register after shifted left
###          $v1 -> the "right" register after shifted left
### Guarantees : Does not alter any registers except for $v0, $v1
shiftLeft2Registers:
    sll $v0, $a0, 1
    srl $v1, $a1, 31
    addu $v0, $v0, $v1
    sll $v1, $a1, 1
    jr $ra
```

4.1.7 Dịch phải chuỗi 64 bit lưu trong 2 thanh ghi

Tên hàm: shiftRight2Registers

- Nhiệm vụ: Dịch phải chuỗi 64 bit lưu trong 2 thanh ghi.
- Input:
  - Thanh ghi $\$a_0$ chứa nửa trái 32 bit của chuỗi.
  - Thanh ghi $\$a_1$ chứa nửa phải 32 bit của chuỗi.
- Output:
  - Thanh ghi $\$v_0$ chứa nửa trái 32 bit của chuỗi sau khi dịch phải 1 đơn vị.
  - Thanh ghi $\$v_1$ chứa nửa phải 32 bit của chuỗi sau khi dịch phải 1 đơn vị.
- Hàm đảm bảo không thay đổi bất kì giá trị của thanh ghi nào trừ thanh ghi $\$v_0$ và $\$v_1$.

```
### Shift right logical the bits across 2 registers
### Input  : $a0 -> the "left" register
###          $a1 -> the "right" register
### Output : $v0 -> the "left" register after shifted right
###          $v1 -> the "right" register after shifted right
### Guarantees : Does not alter any registers except for $v0, $v1
shiftRight2Registers:
   srl $v1, $a1, 1
   sll $v0, $a0, 31
   or $v1, $v0, $v1
   srl $v0, $a0, 1
   jr $ra
```

4.1.8 Cộng 2 số nguyên không dấu và có lưu bit tràn

Tên hàm: addUnsignedWithCarry

- Nhiệm vụ: Thực hiện phép cộng 2 số nguyên không dấu, có lưu bit tràn.
- Input:
  - Thanh ghi $\$a_0$ chứa số nguyên không dấu thứ nhất.
  - Thanh ghi $\$a_1$ chứa số nguyên không dấu thứ hai.
- Output:
  - Thanh ghi $\$v_0$ chứa tổng của 2 số nguyên không dấu.
  - Thanh ghi $\$v_1$ bằng 1 nếu phép cộng bị tràn và 0 nếu ngược lại.
- Hàm đảm bảo không thay đổi bất kì giá trị của thanh ghi nào trừ thanh ghi $\$v_0$ và $\$v_1$.

```
### Add two registers and also returns a carry bit from the 32nd bit
### Input  : $a0 -> the first register
###          $a1 -> the second register
### Output : $v0 -> the sum of the two registers
###        : $v1 -> 1 if the sum overflow, 0 otherwise
### Guarantees: Does not alter any registers except for $v0, $v1
addUnsignedWithCarry:
   addu $v0, $a0, $a1        # $v0 = 32 lower bits of sum
   nor $v1, $a0, $zero       # flip all bits of $a0
   sltu $v1, $v1, $a1        # $v1 = 1 if $a0 + $a1 > 0xFFFF FFFF
   jr $ra
```

4.2 Phép nhân

Tên hàm: multiply

- Nhiệm vụ: Thực hiện phép nhân 2 số thực độ chính xác kép không sử dụng các lệnh với số thực.
- Input:

  - Thanh ghi a_1 chứa 32 bit cao của thừa số thứ nhất.
  - Thanh ghi a_0 chứa 32 bit thấp của thừa số thứ nhất.
  - Thanh ghi a_3 chứa 32 bit cao của thừa số thứ hai.
  - Thanh ghi a_2 chứa 32 bit thấp của thừa số thứ hai.
- Output:
  - Thanh ghi v_1 chứa 32 bit cao của tích.
  - Thanh ghi v_0 chứa 32 bit thấp của tích.
- Giả thiết, cả hai thừa số đều không phải NaN, Infinity và denorms.

```
### Multiply two double-precision floating point number stored in general-purpose registers
### Input  : $a1 -> upper 32 bits of the first double
###          $a0 -> lower 32 bits of the first double
###          $a3 -> upper 32 bits of the second double
###          $a2 -> lower 32 bits of the second double
### Output : $v1 -> upper 32 bits of the resulting double
###          $v0 -> lower 32 bits of the resulting double
### Assumptions: The two double are not NaNs, Infinities or denormalized numbers
multiply:
```

Hàm multiply gồm những bước chính sau:

1. Cấp phát bộ nhớ trên stack và lưu địa chỉ trả về và giá trị các thanh ghi trên stack.

```
    STEP1_SAVE_STATES:
    ## saving return address and registers' states
        addi $sp, $sp, -36
        sw $ra, 0($sp)

        sw $s0, 4($sp)
        sw $s1, 8($sp)
        sw $s2, 12($sp)
        sw $s3, 16($sp)
        sw $s4, 20($sp)
        sw $s5, 24($sp)
        sw $s6, 28($sp)
        sw $s7, 32($sp)
```

2. Trích các trường của hai thừa số.

```
    STEP2_FIELD_ETRACTION:
       ## extracting fields of the multiplicand (first double)
       jal extractSign              # note: does not alter $a0 and $a1
       add $s0, $v0, $zero          # s0 <- sign of the multiplicand

       jal extractBiasedExponent    # note: does not alter $a0 and $a1
       add $s1, $v0, $zero          # s1 <- biased exponent of the
           multiplicand

       jal extractFraction          # note: does not alter $a0 and $a1
```

```
        add $s3, $v1, $zero          # s3 <- the 20 upper bits of the
            multiplicand's fraction
        add $s2, $v0, $zero          # s2 <- the 32 lower bits of the
            multiplicand's fraction

        ## extracting fields of the multiplier (second double)
        add $a1, $a3, $zero
        add $a0, $a2, $zero

        jal extractSign          # note: does not alter $a0 and $a1
        add $s4, $v0, $zero          # $s4 <- sign of the multiplier

        jal extractBiasedExponent        # note: does not alter $a0 and $a1
        add $s5, $v0, $zero          # $s5 <- biased exponent of the
            multiplier

        jal extractFraction          # note: does not alter $a0 and $a1
        add $s7, $v1, $zero          # $s7 <- the 20 upper bits of the
            multiplier's fraction
        add $s6, $v0, $zero          # $s6 <- the 32 lower bits of the
            multiplier's fraction
```

Kiểm tra trường hợp một trong hai thừa số bằng 0. Nếu khác 0, ta cần set thêm bit ngay trước bit cao nhất của phần thập phân để thể hiện việc nó được cộng với 1.0.

```
    ## Zero checking + Add 1.0
       lui $t0, 0x0010           # bit mask 0x0010 0000

       or $t1, $s3, $s2
       or $t1, $t1, $s1          # $t1 = 0 if the multiplicand is 0

       or $t2, $s7, $s6
       or $t2, $t2, $s5          # $t2 = 0 if the multiplier is 0

       bne $t1, $zero, NOT_0_MULTIPLICAND
          ### set to 0
          add $s1, $zero, $zero
          add $s2, $zero, $zero
          add $s3, $zero, $zero
          j STEP7_SIGN_BIT          # set sign bit and return
       NOT_0_MULTIPLICAND:
       or $s3, $s3, $t0          # set the 21st bit to 1 if the multiplicand
           is not zero

       bne $t2, $zero, NOT_0_MULTIPLIER
          ### set to 0
          add $s1, $zero, $zero
          add $s2, $zero, $zero
          add $s3, $zero, $zero
          j STEP7_SIGN_BIT          # set sign bit and return
       NOT_0_MULTIPLIER:
       or $s7, $s7, $t0          # set the 21st bit to 1 if the multiplier is
           not zero
```

3. Tính toán trường mũ (cộng với bias) của tích.

```
    STEP3_BIASED_EXPONENT:
    ## calculating the biased exponent of the result
       sub $s1, $s1, $s5            # s1 <- nonbiased exponent
       addiu $s1, $s1, 0x03FF       # s1 <- biased exponent
```

4. Tính toán trường thập phân của thương (chưa chuẩn hóa và làm tròn).

```
    STEP4_FRACTION:
    ## multiplying the two fractions
       multu $s2, $s6
       mflo $t0                 # lower 32 bit of the product (0 - 31)
       mfhi $t1

       multu $s2, $s7
       mflo $t2
       mfhi $t3

       multu $s3, $s6
       mflo $t4
       mfhi $t5

       multu $s3, $s7
       mflo $t6
       mfhi $t7

       # calculate next 32 bits of the product (32 - 63)
       add $a0, $t1, $zero
       add $a1, $t2, $zero
       jal addUnsignedWithCarry         # note: does not alter any registers
           except for $v0, $v1
       add $t2, $v1, $zero

       add $a0, $v0, $zero
       add $a1, $t4, $zero
       jal addUnsignedWithCarry         # note: does not alter any registers
           except for $v0, $v1
       add $t2, $t2, $v1                # carry from the current 32 bits

       add $t1, $v0, $zero              # next 32 bits of the product

       # calculate next 32 bits of the product (64 - 95)
       add $a0, $t2, $zero
       add $a1, $t3, $zero
       jal addUnsignedWithCarry         # note: does not alter any registers
           except for $v0, $v1
       add $t3, $v1, $zero

       add $a0, $v0, $zero
       add $a1, $t5, $zero
       jal addUnsignedWithCarry         # note: does not alter any registers
           except for $v0, $v1
       add $t3, $v1, $t3

       add $a0, $v0, $zero
       add $a1, $t6, $zero
       jal addUnsignedWithCarry         # note: does not alter any registers
           except for $v0, $v1
       add $t3, $v1, $t3                # carry from the current 32 bits

```

```
48          add $t2, $v0, $zero                 # next 32 bits of the product
49
50          # calculate next 32 bits of the product (96 - 127)
51          add $t3, $t3, $t7                   # no overflow should happen here!
52                                  # last 32 bits of the product
53
54          # $t0 -> 0-31
55          # $t1 -> 32-63
56          # $t2 -> 64-95
57          # $t3 -> 96-127
```

5. Làm tròn và chuẩn hóa.

```
1       STEP5_ROUND_NORMALIZE:
2       ## Rounding and normalize
3          ori $t4, $zero, 0x0200              # bit mask 0x0000 0200
4          and $t4, $t3, $t4                   # $t4 = 0 if the product doesn't need
                normalizing, != 0 otherwise
5
6          beq $t4, $zero, NOT_NORMALIZE
7             sll $s3, $t3, 11
8             srl $t4, $t2, 21
9             or $s3, $s3, $t4
10
11            sll $s2, $t2, 11
12            srl $t4, $t1, 21
13            or $s2, $s2, $t4
14
15            addiu $s1, $s1, 1
16            j EXIT_NORMALIZE
17         NOT_NORMALIZE:
18            sll $s3, $t3, 12
19            srl $t4, $t2, 20
20            or $s3, $s3, $t4
21
22            sll $s2, $t2, 12
23            srl $t4, $t1, 20
24            or $s2, $s2, $t4
25         EXIT_NORMALIZE:
```

6. Kiểm tra tràn số.

```
1       STEP6_UNDERFLOW_OVERFLOW:
2       ## overflow/underflow checking
3          slti $t0, $s1, 0x07FF
4          bne $t0, $zero, EXIT_OVERRFLOW_CHECK
5             ### set to infinity
6             ori $s1, $zero, 0x07FF
7             add $s2, $zero, $zero
8             add $s3, $zero, $zero
9          EXIT_OVERRFLOW_CHECK:
10
11         slti $t0, $s1, 0
12         beq $t0, $zero, EXIT_UNDERFLOW_CHECK
13            ### set to zero
14            add $s1, $zero, $zero
```

```
15          add $s2, $zero, $zero
16          add $s3, $zero, $zero
17      EXIT_UNDERFLOW_CHECK:
```

7. Tính toán bit dấu của tích.

```
1   STEP7_SIGN_BIT:
2   ## determining the sign bit of the result
3       xor $s0, $s0, $s4
```

8. Ghép các trường vào để được tích.

```
1   STEP8_FIELD_COMBINE:
2   ## combining the field
3      ### Fraction field
4      add $v0, $s2, $zero
5
6      lui $t0, 0x000F
7      ori $t0, $t0, 0xFFFF
8      and $s3, $s3, $t0
9      or $v1, $s3, $zero
10
11     ### Exponent field
12     sll $s1, $s1, 20
13     or $v1, $v1, $s1
14
15     ### Sign bit
16     sll $s0, $s0, 31
17     or $v1, $v1, $s0
```

9. Khôi phục địa chỉ trả về và các giá trị của thanh ghi và thoát khỏi hàm.

```
1   STEP9_RETRIEVE_STATE:
2   ## retrieving return address and registers' states
3      lw $ra, 0($sp)
4
5      lw $s0, 4($sp)
6      lw $s1, 8($sp)
7      lw $s2, 12($sp)
8      lw $s3, 16($sp)
9      lw $s4, 20($sp)
10     lw $s5, 24($sp)
11     lw $s6, 28($sp)
12     lw $s7, 32($sp)
13
14     addi $sp, $sp, 36
15
16     jr $ra
```

4.3 Phép chia

Tên hàm: divide

- Nhiệm vụ: Thực hiện phép chia 2 số thực độ chính xác kép không sử dụng các lệnh với số thực.
- Input:
  - Thanh ghi $a_1 chứa 32 bit cao của số bị chia.
  - Thanh ghi $a_0 chứa 32 bit thấp của số bị chia.
  - Thanh ghi $a_3 chứa 32 bit cao của số chia.
  - Thanh ghi $a_2 chứa 32 bit thấp của số chia.
- Output:
  - Thanh ghi $v_1 chứa 32 bit cao của thương.
  - Thanh ghi $v_0 chứa 32 bit thấp của thương.
- Giả thiết, số bị chia và số chia đều không phải NaN, Infinity và denorms.

```
### Divide two double-precision floating point number stored in general-purpose registers
### More specifically, divide the first double by the second double
### Input  : $a1 -> upper 32 bits of the first double
###          $a0 -> lower 32 bits of the first double
###          $a3 -> upper 32 bits of the second double
###          $a2 -> lower 32 bits of the second double
### Output : $v1 -> upper 32 bits of the resulting double
###          $v0 -> lower 32 bits of the resulting double
### Assumptions: The two doubles are not NaNs, Infinities or denormalized numbers
divide:
```

Hàm divide gồm những bước chính sau:

1. Cấp phát bộ nhớ trên stack và lưu địa chỉ trả về và giá trị các thanh ghi trên stack.

```
    STEP1_SAVE_STATES:
    ## saving return address and registers' states
       addi $sp, $sp, -36
       sw $ra, 0($sp)

       sw $s0, 4($sp)
       sw $s1, 8($sp)
       sw $s2, 12($sp)
       sw $s3, 16($sp)
       sw $s4, 20($sp)
       sw $s5, 24($sp)
       sw $s6, 28($sp)
       sw $s7, 32($sp)
```

2. Trích các trường của số bị chia và số chia.

```
    STEP2_FIELD_EXTRACTION:
    ## extracting fields of the dividend (first double)
        jal extractSign             # note: does not alter $a0 and $a1
        add $s0, $v0, $zero         # s0 <- sign of the dividend

        jal extractBiasedExponent   # note: does not alter $a0 and $a1
        add $s1, $v0, $zero         # s1 <- biased exponent of the dividend

        jal extractFraction         # note: does not alter $a0 and $a1
        add $s3, $v1, $zero         # s3 <- the 20 upper bits of the dividend
            's fraction
        add $s2, $v0, $zero         # s2 <- the 32 lower bits of the dividend
            's fraction


    ## extracting fields of the divisor (second double)
        add $a1, $a3, $zero
        add $a0, $a2, $zero

        jal extractSign             # note: does not alter $a0 and $a1
        add $s4, $v0, $zero         # s4 <- sign of the divisor

        jal extractBiasedExponent   # note: does not alter $a0 and $a1
        add $s5, $v0, $zero         # s5 <- biased exponent of the divisor

        jal extractFraction         # note: does not alter $a0 and $a1
        add $s7, $v1, $zero         # s7 <- the 20 upper bits of the divisor'
            s fraction
        add $s6, $v0, $zero         # s6 <- the 32 lower bits of the divisor'
            s fraction
```

Kiểm tra các trường hợp: Cả số bị chia và số chia đều là 0, số bị chia bằng 0 và số chia bằng 0. Nếu khác 0, ta cần set thêm bit ngay trước bit cao nhất của phần thập phân để thể hiện việc nó được cộng với 1.0.

```
    ## Zero checking + Add 1.0
    lui $t0, 0x0010             # bit mask 0x0010 0000

    or $t1, $s3, $s2
    or $t1, $t1, $s1            # $t1 = 0 if the dividend is 0

    or $t2, $s7, $s6
    or $t2, $t2, $s5            # $t2 = 0 if the divisor is 0

    or $t3, $t1, $t2            # $t3 = 0 if both the divisor and
        dividend are 0

    bne $t3, $zero, NOT_0_DIV_0
      ### 0/0 - set to NaN
      ori $s1, $zero, 0x07FF
      addi $s2, $zero, 0xFFFF
      add $s3, $zero, $zero
      j STEP8_FIELD_COMBINE     # return
    NOT_0_DIV_0:

    bne $t1, $zero, NOT_0_DIVIDED
      ### set to 0
      add $s1, $zero, $zero
      add $s2, $zero, $zero
```

```
24        add $s3, $zero, $zero
25        j STEP7_SIGN_BIT              # set sign bit and return
26    NOT_0_DIVIDED:
27    or $s3, $s3, $t0                  # set the 21st bit to 1 if the dividend
          is not zero
28
29    bne $t2, $zero, NOT_DIV_BY_0
30      ### divide by zero - set to infinity
31      ori $s1, $zero, 0x07FF
32      add $s2, $zero, $zero
33      add $s3, $zero, $zero
34      j STEP7_SIGN_BIT              # set sign bit and return
35    NOT_DIV_BY_0:
36    or $s7, $s7, $t0                  # set the 21st bit to 1 if the divisor is
          not zero
```

3. Tính toán trường mũ (cộng với bias) của thương.

```
1     STEP3_BIASED_EXPONENT:
2     ## calculating the biased exponent of the result
3        sub $s1, $s1, $s5              # s1 <- nonbiased exponent
4        addiu $s1, $s1, 0x03FF         # s1 <- biased exponent
```

4. Tính toán trường thập phân của thương (chưa chuẩn hóa và làm tròn).

```
1     STEP4_FRACTION:
2     ## dividing the two fractions
3        add $t0, $zero, $zero          # loop counter
4        addi $t1, $zero, 53            # loop times
5
6        add $t3, $zero, $zero          # used to store the 20 upper bits of the
             result's fraction, also accounts for the bit before the decimal
             point (bit 21)
7        add $t2, $zero, $zero          # used to store the 32 lower bits of the
             result's fraction
8
9        lui $t4, 0x0010                # bit mask 0x0010 0000
10
11       DIVIDE_LOOP:
12            ### Check if the dividend is smaller than the divisor
13            sltu $t5, $s3, $s7
14
15            sltu $t6, $s7, $s3
16            or $t6, $t6, $t5          # $t6 <- 0 if $s3 == $s7, 1 otherwise
17            xori $t6, $t6, 0x0001     # $t6 <- 1 if $s3 == $s7, 0 otherwise
18
19            sltu $t7, $s2, $s6
20            and $t6, $t6, $t7
21
22            or $t5, $t5, $t6          # $t5 = 1 if the dividend is smaller than
                   the divisor
23
24            ### exit on the 54th iteration
25            ### $t5 would be the guard bit on exit
26            beq $t0, $t1, DIVIDE_EXIT
27
```

```
            ### Skip if the dividend is smaller
            bne $t5, $zero, EXIT_SET_BIT
               ### Subtract dividend from the divisor
               sltu $t5, $s2, $s6
               subu $s2, $s2, $s6
               subu $s3, $s3, $s7
               subu $s3, $s3, $t5

               ### Set the bits of the result
               sltiu $t5, $t0, 21   # from the 22nd iteration, set the bits
                   of $t2
               beq $t5, $zero, UPDATE_LOWER
               or $t3, $t3, $t4
               j EXIT_SET_BIT
               UPDATE_LOWER:
               or $t2, $t2, $t4
            EXIT_SET_BIT:

            ### shifting left the bits of the dividend
            add $a0, $s3, $zero
            add $a1, $s2, $zero
            jal shiftLeft2Registers # note: does not alter any registers
                except for $v0, $v1
            add $s3, $v0, $zero
            add $s2, $v1, $zero

            ### Circularly shifting right bit mask
            srl $t4, $t4, 1
            bne $t4, $zero, EXIT_UPDATE_MASK
               lui $t4, 0x8000      # refresh mask to 0x8000 0000
            EXIT_UPDATE_MASK:

            addi $t0, $t0, 1
            j DIVIDE_LOOP

      DIVIDE_EXIT:

      add $s3, $t3, $zero          # storing the upper 20 bits of the
          fraction, also accounting for the bit before the decimal point (
          bit 21)
      add $s2, $t2, $zero          # storing the lower 32 bits of the
          fraction
```

5. Làm tròn và chuẩn hóa.

```
   STEP5_ROUND_NORMALIZE:
   ## Rounding and normalize
      # $t5 now acts like the guard bit
      srl $t4, $s3, 20             # $t4 = 1 if the number is normalized
      bne $t4, $zero, EXIT_NORMALIZE
         ### shift left the fraction of the result
         add $a0, $s3, $zero
         add $a1, $s2, $zero
         jal shiftLeft2Registers   # Note: does not alter any registers
             except for $v0, $v1
         add $s3, $v0, $zero
         add $s2, $v1, $zero

         addu $s2, $s2, $t5        # add the sticky bit to the fraction of
             the result
```

```
14
15          addiu $s1, $s1, -1        # decrement the exponent of the result
16      EXIT_NORMALIZE:
```

6. Kiểm tra tràn số.

```
1       STEP6_UNDERFLOW_OVERFLOW:
2       ## overflow/underflow checking
3          slti $t0, $s1, 0x07FF
4          bne $t0, $zero, EXIT_OVERRFLOW_CHECK
5             ### set to infinity
6             ori $s1, $zero, 0x07FF
7             add $s2, $zero, $zero
8             add $s3, $zero, $zero
9          EXIT_OVERRFLOW_CHECK:
10
11         slti $t0, $s1, 0
12         beq $t0, $zero, EXIT_UNDERFLOW_CHECK
13            ### set to zero
14            add $s1, $zero, $zero
15            add $s2, $zero, $zero
16            add $s3, $zero, $zero
17         EXIT_UNDERFLOW_CHECK:
```

7. Tính toán bit dấu của thương.

```
1       STEP7_SIGN_BIT:
2       ## determining the sign bit of the result
3           xor $s0, $s0, $s4
```

8. Ghép các trường vào để được thương.

```
1       STEP8_FIELD_COMBINE:
2       ## combining the field
3          ### Fraction field
4          add $v0, $s2, $zero
5
6          lui $t0, 0x000F
7          ori $t0, $t0, 0xFFFF
8          and $s3, $s3, $t0
9          or $v1, $s3, $zero
10
11         ### Exponent field
12         sll $s1, $s1, 20
13         or $v1, $v1, $s1
14
15         ### Sign bit
16         sll $s0, $s0, 31
17         or $v1, $v1, $s0
```

9. Khôi phục địa chỉ trả về và các giá trị của thanh ghi và thoát khỏi hàm.

```
STEP9_RETRIEVE_STATE:
## retrieving return address and registers' states
    lw $ra, 0($sp)

    lw $s0, 4($sp)
    lw $s1, 8($sp)
    lw $s2, 12($sp)
    lw $s3, 16($sp)
    lw $s4, 20($sp)
    lw $s5, 24($sp)
    lw $s6, 28($sp)
    lw $s7, 32($sp)

    addi $sp, $sp, 36

    jr $ra
```

5 Thống kê

Các nhóm lệnh sử dụng gồm có: Lệnh R, lệnh I, lệnh J và lệnh Coprocessor (mtc1, mfc1).

Tên hàm	Số lệnh R	Số lệnh I	Số lệnh J	Số lệnh Coproc	Tổng số lệnh
readDouble	2	1	0	1	**4**
writeDouble	2	1	0	1	**4**
extractSign	2	0	0	0	**2**
extractBiasedExponent	3	1	0	0	**4**
extractFraction	3	2	0	0	**5**
shiftLeft2Registers	5	0	0	0	**5**
shiftRight2Registers	5	0	0	0	**5**
multiply	80	34	14	0	**128**
divide	66	47	13	0	**126**

Bảng 1: Thống kê số lệnh của các hàm

6 Kiểm thử

Thời gian thực thi của chương trình sẽ được tính, với giả định rằng CPI của tất cả các lệnh đều bằng 1, tần số clock là 3.4 GHz.

Ta có công thức đơn giản sau:

$$\text{Execution Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Frequency}} = \frac{\text{Instruction Count}}{3.4} (\text{ns})$$

6.1 Phép nhân

STT	Thừa số 1	Thừa số 2	Kết quả	R	I	J	Tổng lệnh	Thời gian (ns)
1	1.0	1.0	1.0	99	39	12	150	44.12
2	0.0	0.0	0.0	41	30	8	79	23.24
3	0.35	-0.0	-0.0	42	31	8	81	23.82
4	0.5	0.25	0.125	99	39	12	150	44.12
5	1E-200	4E-200	0.0	99	39	12	150	44.12
6	2.3E-200	-1.2E-150	-0.0	99	39	12	150	44.12
7	1.2	-1.2	-1.439...9	99	39	12	150	44.12
8	4.5E200	12E139	Infinity	101	40	12	153	45
9	-2.11E200	4.23E167	-Infinity	99	39	12	150	44.12
10	-0.0	3E289	-0.0	41	30	8	79	23.24
11	0.3	-0.3	-0.089...9	99	39	12	150	44.12
12	0.12	12	1.44	99	39	12	150	44.12
13	0.00001	0.00000001	1.0E-13	99	39	12	150	44.12
14	200E-10	10E10	2000.0	99	39	12	150	44.12
15	-2.2	1.1	-2.420...04	99	39	12	150	44.12
16	-1.1	-1.1	1.210...02	99	39	12	150	44.12
17	-0.0	-1.23	0.0	41	30	8	79	23.24
18	2E40	4E-20	7.9...9E20	99	39	12	150	44.12
19	-0.9	-0.2	0.18	99	39	12	150	44.12
20	0.23	-2	-0.46	99	39	12	150	44.12
21	1.54	0.5	0.77	99	39	12	150	44.12
22	-12	-0.1	1.2	99	39	12	150	44.12
23	17	0.3	5.1	99	39	12	150	44.12
24	6	6	36.0	99	39	12	150	44.12
25	0.1	0.1	0.01	99	39	12	150	44.12
26	-12222	0.1	-1222.2	99	39	12	150	44.12
27	7.8	-0.25	1.95	99	39	12	150	44.12
28	-9	3E-10	-2.69...9E-9	99	39	12	150	44.12
29	3	-0.33	-0.99	99	39	12	150	44.12
30	15	5	75.0	99	39	12	150	44.12

Bảng 2: Bảng thống kê số lệnh mỗi loại và thời gian thực thi các test case của hàm multiply

6.2 Phép chia

STT	Số bị chia	Số chia	Kết quả	R	I	J	Tổng lệnh	Thời gian (ns)
1	1	1	1.0	907	311	114	1332	391.765
2	4	2	2.0	907	311	114	1332	391.765
3	5	2	2.5	912	313	115	1340	394.118
4	5.6	7	0.7999...8	1042	362	124	1528	449.412
5	10	3	3.3333...5	1047	364	125	1536	451.765
6	+0	+0	NaN	40	33	8	81	23.82
7	−0	−0	NaN	40	33	8	81	23.82
8	+0	−0	NaN	40	33	8	81	23.82
9	−0	+0	NaN	40	33	8	81	23.82
10	1E−10	−0	−Infinity	42	33	8	83	24.412
11	−2E100	1E−300	−Infinity	1084	379	126	1589	467.353
12	0	5E−200	0.0	42	31	8	81	23.82
13	−2.5	5	−0.5	907	311	114	1332	391.765
14	23	−5	−4.6	1032	361	123	1516	445.882
15	0.1	−12	−0.00833...3	972	337	119	1428	420
16	−5	−3	1.6666...7	1047	364	125	1536	451.765
17	17E−300	1E300	0.0	1045	362	127	1534	451.176
18	−3E−200	2E200	−0.0	1076	374	127	1576	463.529
19	1.76	2	0.88	1032	361	124	1517	446.176
20	−2	−2	1.0	907	311	114	1332	391.765
21	8E10	2E5	400000.0...6	942	322	120	1384	407.059
22	9.5	2.5	3.8000...3	1052	366	126	1544	454.118
23	−0.001	7	−0.00142857...	1047	364	126	1537	452.059
24	1	8	0.125	907	311	114	1332	391.765
25	1	0.125	8.0	907	311	114	1332	391.765
26	0.42	−0.7	−0.6	1037	363	124	1524	448.235
27	0.81	−0.9	−0.8999...9	1047	364	125	1536	451.765
28	−3.14	1.57	−2.0	907	311	114	1332	391.765
29	4.2	0.14	29.9999...3	1167	412	133	1712	503.529
30	11.99	1100	0.0109	1027	359	125	1511	444.412

Bảng 3: Bảng thống kê số lệnh mỗi loại và thời gian thực thi các test case của hàm divide