BÁO CÁO BÀI TẬP

**Môn học: Lập trình hệ thống**

**Buổi báo cáo: Lab 1**

**Tên chủ đề:** Data Lab: Tính toán Bits

*GVHD: Đỗ Thị Hương Lan*

*Ngày thực hiện: 29/09/2025*

**THÔNG TIN CHUNG:**

*(Liệt kê tất cả các thành viên trong nhóm)*

Lớp: NT209.Q12.ANTT

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Họ và tên** | **MSSV** | **Email** |
| 1 | Trần Ngọc Danh | 22520200 |  |
| 2 | Đoàn Minh Khang | 22520609 |  |
| 3 | Phạm Huỳnh Tấn Khang | 22520624 |  |

1. **ĐÁNH GIÁ KHÁC:**

|  |  |
| --- | --- |
| **Nội dung** | **Kết quả** |
| Tổng thời gian thực hiện bài thực hành trung bình |  |
| Link Video thực hiện  *(nếu có)* |  |
| Ý kiến *(nếu có)*  + Khó khăn  + Đề xuất … |  |
| Điểm tự đánh giá |  |

**Phần bên dưới của báo cáo này là báo cáo chi tiết của nhóm thực hiện.**

BÁO CÁO CHI TIẾT

1. **Tính toán bit đơn giản**
   1. Hàm bitOr(x, y)

Hàm sử dụng định luật De Morgan để tính toán phép OR chỉ với toán tử ~ và &

Định lý De Morgan được phát biểu như sau:



Nên ta suy ra hàm:

1. int bitOr(int x, int y)

2. {

3.     return ~(~x & ~y);

4. }

5.

* 1. Hàm negative(x)

Hàm dựa trên biểu diễn **bù 2** trong hệ nhị phân để tính số đối. Cụ thể, ~x + 1 cho ta giá trị nghịch đảo của x dẫn đến cho ta kết quả là số đối của x.

Lưu số âm bằng bù 2. Lấy tất cả bit đảo lại rồi cộng thêm 1 sẽ cho ra giá trị âm tương ứng

1. int negative(int x)

2. {

3. return ~x + 1; // dùng bù 2

4. }

5.

* 1. Hàm getHexcha(x, n)

Mỗi chữ số trong hệ 16 chiếm đúng 4 bit. Khi ta muốn lấy chữ số thứ n, chỉ cần dịch sang phải để nó nằm ở cuối, rồi dùng mặt nạ để loại bỏ các bit thừa

Phương pháp: Dịch số x sang phải n\*4 bit để đưa nhóm 4 bit cần lấy về cuối, rồi AND với 0xF để giữ lại đúng 4 bit đó

1. int getHexcha(int x, int n)

2. {

3. int shiftAmount = n << 2; // Tính số bit cần dịch: n \* 4, tương đương n << 2

4. // Dịch phải x để đưa byte thứ n về vị trí thấp nhất

5. int result = x >> shiftAmount;

6. // Dùng mask 0xF để xóa tất cả các bit trừ 8 bit cuối cùng

7. return result & 0xF;

8. }

9.

* 1. Hàm flipByte(x, n)
  2. Hàm divpw2(x, n)

1. **Các phép kiểm tra dựa trên tính toán bit**
   1. Hàm isEqual(x, y)

Dùng XOR để kiểm tra: nếu x ^ y = 0 thì hai số bằng nhau. Sau đó phủ định kết quả bằng !

XOR của hai số bằng 0 nghĩa là từng bit của chúng giống nhau. Nhờ vậy, ta xác định được hai số có bằng nhau mà không cần so sánh trực tiếp

1. int isEqual(int x, int y)

2. {

3.     return !(x ^ y);

4. }

5.

* 1. Hàm is16x(x)

**Ý tưởng**: Các số chia hết cho 16 như 16, 32, 64,… khi viết dưới dạng số nhị phân đều có 4 bit cuối là 0. Do đó, ta chỉ việc kiểm tra 4 số cuối của số đầu vào, nếu cả 4 bit bằng 0 thì số đó chia hết cho 16.

Phương pháp được sử dụng là sử dụng mask 15 có 4 bit cuối ở dạng nhị phân là 1111 để lấy ra 4 bit cuối của x và gán vào biến last\_4\_bits.

Sau đó, sử dụng toán tử (!) để kiểm tra, nếu last\_4\_bits = 0 thì !last\_4\_bits = 1 (chia hết cho 16), ngược lại !last\_4\_bits = 0. Dùng & 1 để đảm bảo kết quả trả về là 0 hoặc 1.

1. int is16x(int x)

2. {

3.     *// Vì 16 = 2^4 nên ta chỉ cần kiểm tra 4 bit cuối của x có bằng 0 hay không*

*4.     // Nếu 4 bit cuối của x = 0 thì x chia hết cho 16*

*5.     // 15 = 0b0000...00001111, dùng để lấy 4 bit cuối của x*

*6.     // Ta có thể dùng phép AND (&) để lấy 4 bit cuối của x;*

7.     int last\_4\_bits = 0;

8.     last\_4\_bits += x & 15;

9.

10.     *// Dùng toán tử ! kiểm tra.*

*11.     // Nếu last\_4\_bits = 0 thì !last\_4\_bits = 1 (chia hết cho 16), ngược lại !last\_4\_bits = 0. Dùng & 1 để lấy bit cuối cùng*

12.     return (!last\_4\_bits) & 1;

13. }

14.

* 1. Hàm isPositive(x)

**Ý tưởng**: Bit dấu trong bù 2 cho biết số âm hay dương, còn (x | -x) khai thác tính chất bù 2 để phân biệt 0 với số khác 0. Nhờ vậy hàm trả về chính xác 1 khi x dương và 0 trong các trường hợp còn lại

Dịch phải 31 bit để lấy bit dấu. Nếu kết quả = 0 thì số không âm, đảo lại bằng ! để được NotNegative

Dùng (x | -x) >> 31 & 1 để kiểm tra khác 0: kết quả = 1 nếu x ≠ 0, ngược lại = 0 ( vì bù 2 của 0 vẫn bằng 0 )

Kết hợp hai điều kiện bằng & để ta có kết quả bằng 1 khi x > 0

1. int isPositive(int x)

2. {

3.     *// Điều kiện 1: x không phải là số âm (tức là x >= 0).*

4.     int NotNegative = !(x >> 31);

5.     *// Điều kiện 2: x không bằng 0.*

6.     int NotZero = ((x | -x) >> 31) & 1;

7.     return NotNegative & NotZero;

8. }

* 1. isGE2n(x, n)

**Ý tưởng:** Tính 2n bằng cách lấy 1 và dịch trái n lần. Sau đó tính x – 2n bằng cách lấy x cộng với bù 2 của 2n rồi cộng với 1.

Nếu x ≥ 2n thì bit dấu của (x – 2n) = 0, ngược lại nếu x < 2n thì bit dấu của (x - 2n) = 1. Vậy nên ta sẽ lấy bit dấu của (x – 2n) bằng cách dịch phải 31 bit và & 1 để đảm bảo kết quả thu được là 0 hoặc 1.

Cuối cùng, dùng toán tử (~) để đảo ngược bit dấu. Nếu bit dấu là 0 (x - 2n ≥ 0) thì trả về 1, còn nếu bit dấu là 1 (x - 2n < 0) thì trả về 0.

1. int isGE2n(int x, int y)

2. {

3.     int exp2\_n = 1 << y; // Tính 2^n bằng cách lấy 1 dịch trái y lần

4.     int diff = x + ~(exp2\_n) + 1; // // Tính x - 2^n bằng cách dùng công thức a - b = a + ~b + 1

5.

6.     // Nếu x >= 2^n thì bit dấu của (x - 2^n) = 0, ngược lại nếu x < 2^n thì bit dấu của (x - 2^n) = 1

7.     // Lấy bit dấu của (x - 2^n) bằng cách dịch phải 31 bit và & 1

8.     int sign\_bit = (diff >> 31) & 1;

9.

10.     // Ta cần trả về 1 khi x >= 2^n (sign\_bit = 0), ngược lại trả về 0 khi x < 2^n (sign\_bit = 1)

11.     // Nên ta cần đảo bit dấu bằng NOT (~) và & 1

12.     return ~sign\_bit & 1;

13. }

14.