WRITE-UP “Bitwise Civilization”

Nama : M. Rayhan Farrukh
NIM : 13523035
Kelas : K01

1. chicken_or_beef

a. Solusi

```
int chicken_or_beef(int chicken, int beef) {
    return (beef << 1) & 0x0f | (chicken >> 4) & 0x0f;
}
```

b. Penjelasan

Pertama mengalikan beef dengan 2 (beef << 1) lalu untuk mengambil 4 bit pertama, bit selain 4 bit pertama harus dimatikan yaitu dengan meng-AND-kan dengan 15 (biner 1111 atau hex 0x0f). Begitu pula dengan chicken, setelah mengambil 4 bit kedua, dan memosisikannya pada letak 4 bit pertama (chicken >> 4), di AND-kan juga dengan 15. Kemudian gabungkan hasilnya dengan bitwise OR.

c. Referensi

Algorithms: Bit Manipulation (youtube.com)

2. Masquerade

a. Solusi

```
int masquerade() {
    return (1 << 31) ^ 1;
}
```

b. Penjelasan

Angka terkecil didapat dari 1 << 31 (-2147483648), kemudian di XOR kan dengan 1 yang di kasus ini karena binary representation memiliki lsb 0, maka sama saja dengan menambah dengan 1, sehingga menjadi -2147483647, yaitu angka terkecil kedua di representasi integer.

c. Referensi

Algorithms: Bit Manipulation (youtube.com)

3. airani_iofifteen

a. Solusi

```
int airani_iofifteen(int iofi){
    return !(iofi >> 4 ) & (iofi >> 3 & 1) & (iofi >> 2 & 1) & (iofi >> 1 &
1) & (iofi >> 0 & 1) ;
}
```

b. Penjelasan

Mengecek setiap bit secara manual (1111). Yang pertama dilakukan adalah memastikan bahwa setiap bit diatas bit keempat adalah 0, dengan cara menyingkirkan 4 bit pertama (iofi >> 4), lalu menggunakan operator !. Jika ada bit bernilai 1, operator ! akan menghasilkan 0 sehingga ketika di-AND-kan akan membuat semua hasil menjadi 0. Sisanya menggunakan cara serupa yaitu menggeser bit yang ingin dicek, tetapi setelahnya di-AND-kan dengan 1 untuk mengecek apakah bit tersebut sama dengan 1.

c. Referensi

Algorithms: Bit Manipulation (youtube.com)
Bitwise manipulation

4. yobanashi_deceive

a. Solusi

```
unsigned yobanashi_deceive(unsigned f){
    return f >> 3;
}
```

b. Penjelasan

Karena f memiliki format 0 bits untuk mantissa dan 32 bit untuk *exponent*, maka operasi pada f sama saja dengan operasi pada *exponent*-nya, dan operasi sqrt 3 kali sama dengan membagi *exponent* dengan 2 sebanyak 3 kali, yaitu f >> 3.

c. Referensi

Tidak ada

5. snow_mix

a. Solusi

```
int snow_mix(int n){
    int B = (1 << 23);

    int sum = n ^ B;
    int carry = (n & B) << 1;

    sum = sum ^ carry;

    return sum;
}
```

b. Penjelasan

Menghasilkan n + 2^23. Pertama mendapatkan nilai 2^23 dan disimpan ke variabel B, kemudian menjumlahkan n dengan B menggunakan operator XOR, hasil penjumlahan belum memperhitungkan *carryy*-nya, yaitu bit yang berpindah ke posisi lebih besar karena hasil penjumlahan melebihi basis (hasil 1 XOR 1). Cara menghitung *carry* adalah dengan mencari bit-bit dari n & B yang sama-sama bernilai satu, lalu menggeser ke kanan untuk *column transfer*. Lalu jumlahkan hasil penjumlahan sebelumnya dengan *carry* (sum XOR carry).

c. Referensi

Add two numbers without using arithmetic operators - GeeksforGeeks

6. Sky_hundred

a. Solusi

```
int sky_hundred(int N){
    int is_negative = N >> 31;


    int mod = N & 3;

    int res_0 = N;
```

```
    int res_1 = 1;
    int res_2 = N + 1;
    int res_3 = 0;

    int mask_0 = ((!(mod ^ 0)) << 31) >>31;
    int mask_1 = ((!(mod ^ 1))<< 31) >> 31;
    int mask_2 = ((!(mod ^ 2)) << 31) >> 31;
    int mask_3 = ((!(mod ^ 3)) << 31) >> 31 ;


    int result = (res_0 & mask_0) | (res_1 & mask_1 ) | (res_2 & mask_2 ) |
(res_3 & mask_3);
    printf("%d\n", result);

    return result & ~is_negative;  // Clears the result if N is negative
}
```

b. Penjelasan

Say x = n%4. The XOR value depends on the value if x. If

- *x = 0, then the answer is n.*
- *x = 1, then answer is 1.*
- *x = 2, then answer is n+1.*
- *x = 3, then answer is 0.*

Jadi, hasil akhir akan bergantung pada hasil dari N mod 4. Pertama N mod 4 didapatkan dari variabel **mod**, meng-AND-kan dengan 3 (bit 0011) akan menghasilkan angka sama dengan operasi mod 4. Kemudian nilai akhir dari masing-masing kasus disimpan pada variabel **res.** Lalu cek kasus mana yang terjadi dengan mengecek hasil mod 4. Cara mengeceknya adalah dengan meng-XOR-kan **mod** dengan masing-masing kasus hasil mod 4 (XOR dua angka sama menghasilkan 0) kemudian digunakan operasi !, jika hasil XOR 0 maka operasi ! mengembalikan 1 dan akan mengembalikan 0 ketika hasil XOR tidak nol.

Ide akhirnya adalah mematikan nilai **res** jika kasus tersebut tidak terjadi, yaitu dengan cara meng-AND-kan dengan **mask**-nya. Saat kasus benar, agar ketika di-AND-kan menghasilkan **res** harus di AND kan dengan -1 (bit 1 semua). Sehingga hasil operasi ! tadi harus di jadikan msb (<< 31) kemudian di *sign extend* (>> 31)agar menjadikan semua bit 1 pada kasus hasil operasi ! sama dengan 1. Lalu hasil akhirnya di-AND-kan dengan negasi dari variabel yang mengambil msb (tanda).

c. Referensi

https://www.geeksforgeeks.org/bits-manipulation-important-tactics/

7. ganganji

a. Solusi

```
int ganganji(int x) {
  int result = x + (x >> 3);
  int overflow = ~(1 << 31);
  int msb = ((result >> 31) & 1);
  int status_overflow = msb << 31 >> 31;

  return (~status_overflow & result) | (status_overflow & overflow);
}
```

b. Penjelasan

Mengalikan x dengan 1.125 sama saja dengan mengalikan x dengan (1 + 0.125) sehingga menjadi x + 0.125x atau x + x/8 (x + x >> 3). Lalu variabel '*overflow*' digunakan untuk menyimpan nilai 2147483647. Variabel msb digunakan untuk melihat apakah terjadi *overflow* di hasil perkalian dengan melihat apakah *most significant bit*-nya sama dengan 1. Jika msb sama dengan 1 maka terjadi *overflow*, jika sama dengan 0, maka tidak ada *overflow*.

c. Referensi

Tidak ada

8. kitsch

a. Solusi

```
int kitsch(int x){
    int big = x & (~63);
    int small = x & 63;
    int isNeg = (x >> 31);



    int negSmall = ~small + 1;
    int isneg64multiple = ((small | negSmall) >> 31) & 1;
    int rounding = isNeg & isneg64multiple;
```

```
    int smallMul = (small << 4) + small;
    int smallDiv =  (smallMul >> 6) + rounding;

    int bigDiv = (big >> 6);
    int bigDiv = (bigDiv << 4) + bigDiv;
    return bigDiv + smallDiv;
}
```

b. Penjelasan

Untuk mencegah overflow, maka perhitungan dipisah menjadi dua, perhitungan 6 bit pertama dan 26 bit kedua. Untuk 6 bit pertama akan dikali 17 terlebih dahulu (variabel **smallMul**) lalu dibagi 64 (variabel **smallDiv**). Dan untuk 26 bit kedua dibagi terlebih dahulu kemudian dikali. Lalu untuk meng-*handle* pembulatannya, untuk kasus x positif pembulatannya otomatis di-*handle* saat operasi *shift*, namun untuk negatif harus di-*handle*.

isneg64multiple berfungsi untuk mengecek apakah angka merupakan angka negatif kelipatan 64, yang dimana angka negatif kelipatan 64 akan memiliki 6 bit pertama yang semua nol. Jika x merupakan angka negatif kelipatan 64, maka pembulatan tidak perlu dilakukan sehingga **isneg64multiple** bernilai 0, dan sebaliknya akan bernilai 1. **Isneg64multiple** didapatkan dengan meng-OR-kan **small** dengan angka negatif dari **small (negSmall),** dimana jika x negatif kelipatan 64, maka **~small** akan menjadi -1 (bit 1 semua) dan ketika ditambah satu **negSmall** akan menjadi 0. Ketika **negSmall 0** hasil OR tetap akan memiliki msb 0 sehingga ketika di shift kanan 31 dan di-AND-kan dengan 1 akan menghasilkan 0.

Setelah itu angka rounding dihitung dengan memastikan bahwa angka negatif (**isNeg**) di-AND-kan dengan **isneg64multiple.** Jika **isneg64multiple** menghasilkan 0 maka rounding menjadi 0 sehingga ketika dijumlahkan tidak memengaruhi hasil akhir. Begitu pula sebaliknya, maka hasil penjumlahan akan menyebabkan hasil di rounding menuju 0.

c. Referensi

https://chatgpt.com/share/671ce94a-b108-800c-9983-cc8c2774f26