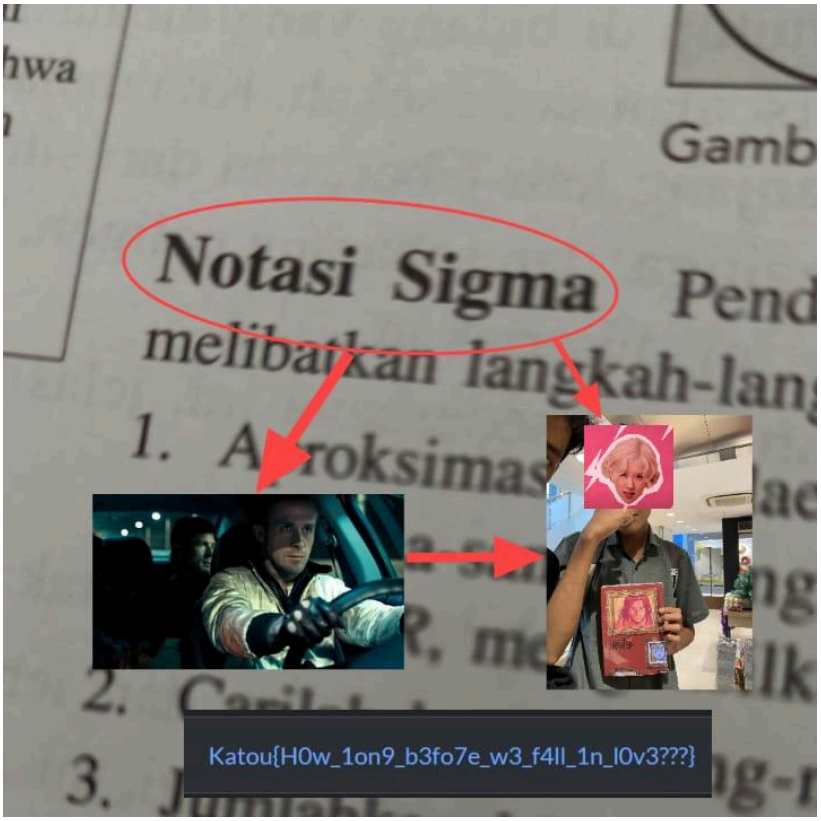
WRITE UP IF-1230
Organisasi dan Arsitektur Komputer

Praktikum 0x1 : Bitwise Civilization



Disusun oleh :

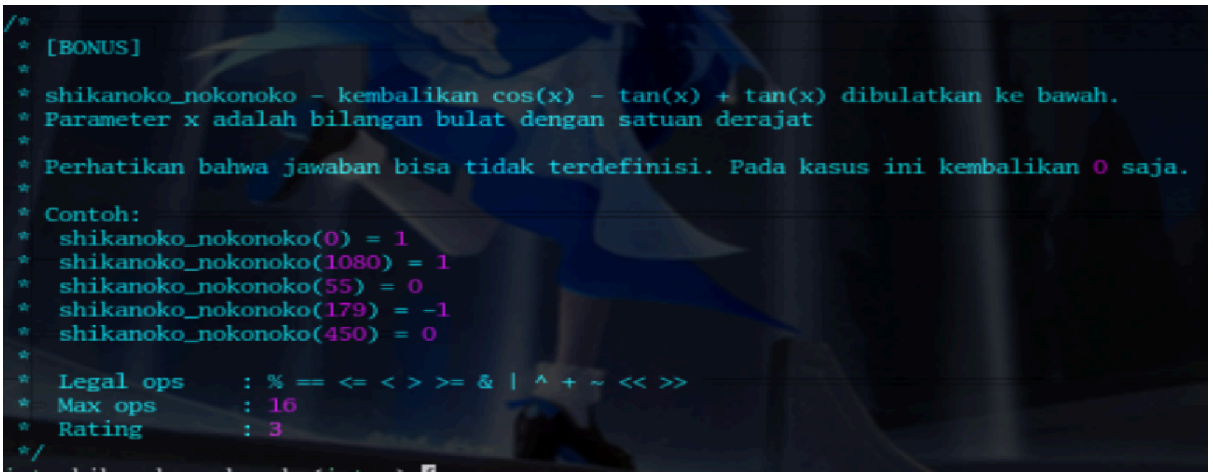
Nayaka “Katounasai” Ghana Subrata (13523090)
IF’23 (K-2)

DAFTAR ISI

SOAL BONUS (Konversi WU)

Shikanoko Nokonoko

- Rating 3 [3 pts : 3 correct]



```
/*
 * [BONUS]
 *
 * shikanoko_nokonoko - kembalikan cos(x) - tan(x) + tan(x) dibulatkan ke bawah.
 * Parameter x adalah bilangan bulat dengan satuan derajat
 *
 * Perhatikan bahwa jawaban bisa tidak terdefinisi. Pada kasus ini kembalikan 0 saja.
 *
 * Contoh:
 *   shikanoko_nokonoko(0) = 1
 *   shikanoko_nokonoko(1080) = 1
 *   shikanoko_nokonoko(55) = 0
 *   shikanoko_nokonoko(179) = -1
 *   shikanoko_nokonoko(450) = 0
 *
 *   Legal ops     : % == <= < > >= & | ^ + ~ << >>
 *   Max ops       : 16
 *   Rating        : 3
 */
```

legal ops : %, ==, <=, <, >, >=, &, |, ^, +, ~, <<, >>
max ops : 16

Solving steps

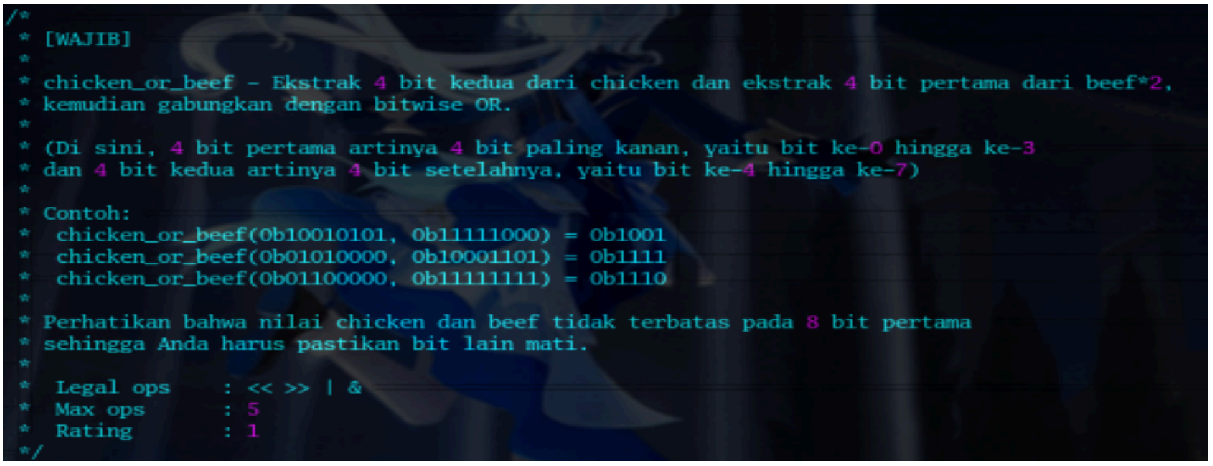
```c
int shikanoko_nokonoko(int x) {
    int mod360 = 180 << 1;
    int eval;

    x = (x % mod360 + mod360) % mod360;
    eval = (x >= 91) & (x <= (100 + 169));

    return (x == 0) | (eval << 31 >> 31);
}
```

Soal Bonus yang Dikerjakan	Soal Wajib yang Dihapus dari Writeup
shikanoko_nokonoko	how_to_sekai_seifuku

SOAL WAJIB

Chicken or Beef

- Rating 1 [3 pts : 1 correct + 2 perfect]



legal ops : <<, >>, | ,&
max ops : 5

Solving steps

```
int chicken_or_beef(int chicken, int beef) {
    return ((chicken >> 4) & 15) | ((beef << 1) & 15);
}
```

Penjelasan

Pada soal ini, aku diharuskan untuk mengambil 4 bit kedua dari chicken dan 4 bit pertama dari beef*2.

Pertama, mengambil 4 bit kedua dari chicken, yakni bit 4 sampai bit ke 7, agar dapat mengekstraknya, terlebih dahulu bit tersebut di shift ke kanan agar letaknya bisa di paling kanan, contoh, bit “0b10010101”, di shift ke kanan menjadi “0b00001001”. Setelah menggeser ke kanan, kita lakukan operasi *and* (&) dengan

bit “0b1111”, atau dalam bentuk desimal adalah 15, untuk chicken, didapatkan bit “0b1001”.

Kedua, mengambil 4 bit pertama dari beef*2. Karena dikali 2, maka beef terlebih dahulu di shift ke kiri sebanyak 1 (shift ke kiri juga dapat didefinisikan sebagai “x << m” menjadi “x * 2^m”. Setelahnya, sama seperti chicken, proses ekstraksi 4 bit beef dilakukan dengan operasi *and* (&) terhadap bit “0b1111”, atau dalam bentuk desimal sama dengan 15.

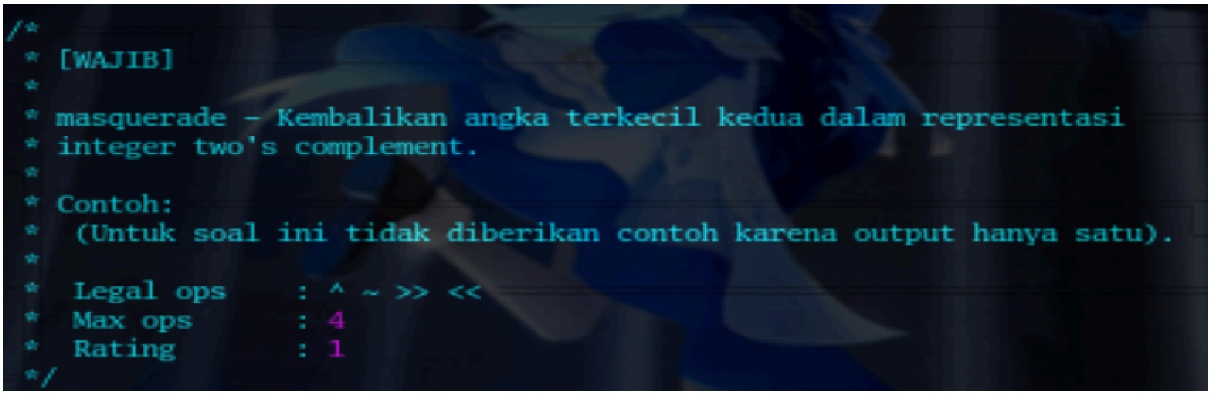
Terakhir, karena sudah mendapatkan masing-masing bit, tinggal digabung dengan operasi *or* (|). Dan soal ini berhasil AC untuk setiap *test case*.

Referensi

Bagi yang mau baca :
https://math.stackexchange.com/questions/1610667/why-shifting-left-1-bit-is-the-same-as-multiply-the-number-by-2

Masquarade

- Rating 1 [3 pts : 1 correct + 2 perfect]



legal ops : ^, ~, << ,>>
max ops : 4

Solving steps

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

Penjelasan

Pada soal ini, kita diharuskan untuk *return* angka terkecil kedua dalam representasi integer two's complement. Angka terkecil dalam two's complement direpresentasikan dengan -2^(N-1), dengan N adalah jumlah bit. Karena sistem 32 bit, maka angka terkecilnya adalah -2^(31), atau sama dengan -2147483648. Dapat disimpulkan bahwa angka terkecil kedua adalah -2147483647.

Kita bisa mendapatkan nilai INT_MIN (-2147483648), dari operasi "1 << 31" (silahkan merefer ke referensi). Selanjutnya, kita gunakan operasi xor untuk mendapatkan nilai terkecil keduanya. Dan ya.. Soal ini berhasil di-ACin.

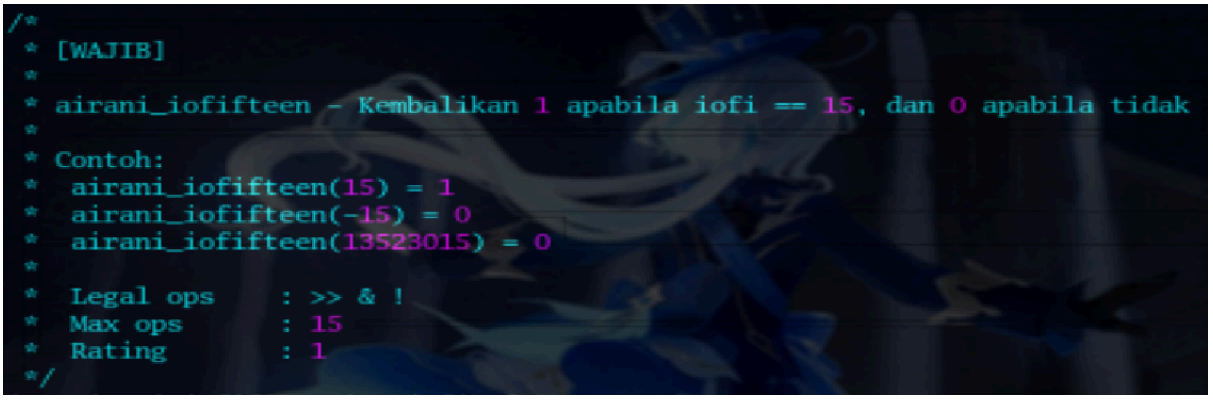
Referensi

Bagi yang mau baca :
https://www.quora.com/What-is-the-smallest-negative-number-using-twos-complement-representation-in-a-5-bit-number

https://pvs-studio.com/en/blog/lessons/0011/

Airani Iofifteen

- Rating 1 [3 pts : 1 correct + 2 perfect]



legal ops : >>, &, !
max ops : 15

Solving steps

```
int airani_iofifteen(int iofi) {
    int lower = iofi & 15;
    int upper = iofi >> 4;

    int isLow = (lower & 8) >> 3 & (lower & 4) >> 2 & (lower &
2) >> 1 & (lower & 1);
    int isUp = !upper;

    return isLow & isUp;
}
```

Penjelasan

Pada soal ini, kita diharuskan untuk *return* 1 jika nilai iofi = 15, dan *return* 0 jika tidak. Idenya adalah, 15 dalam representasi bit adalah “0x1111”, jadi cukup ambil 4 bit dan compare dengan 4 bit lainnya. Jadi, aku mengambil 4 bit pertama dari iofi, dengan cara melakukan operasi *and* (&) dengan 15 (atau dalam representasi bitnya adalah 0x1111). Selanjutnya, shift iofi sebanyak 4 ke kanan (kita sudah menyimpan 4 bit pertama di dalam lower), lalu kita ekstrak 4 bit pertama (yang ada di dalam lower) 1 per satu. Pertama, lower *and* 8 lalu di shift ke kanan 3, ini sama artinya dengan jika kita memiliki bit “0x1000”, maka program akan mengambil “1”, lalu yang kedua mengekstrak bit ketiga dari kanan, dan seterusnya

sampai habis. Setelahnya, lakukan operasi *and* pada masing-masing bit yang telah diekstrak. Jika hasilnya 15, maka isLow akan menghasilkan 1.

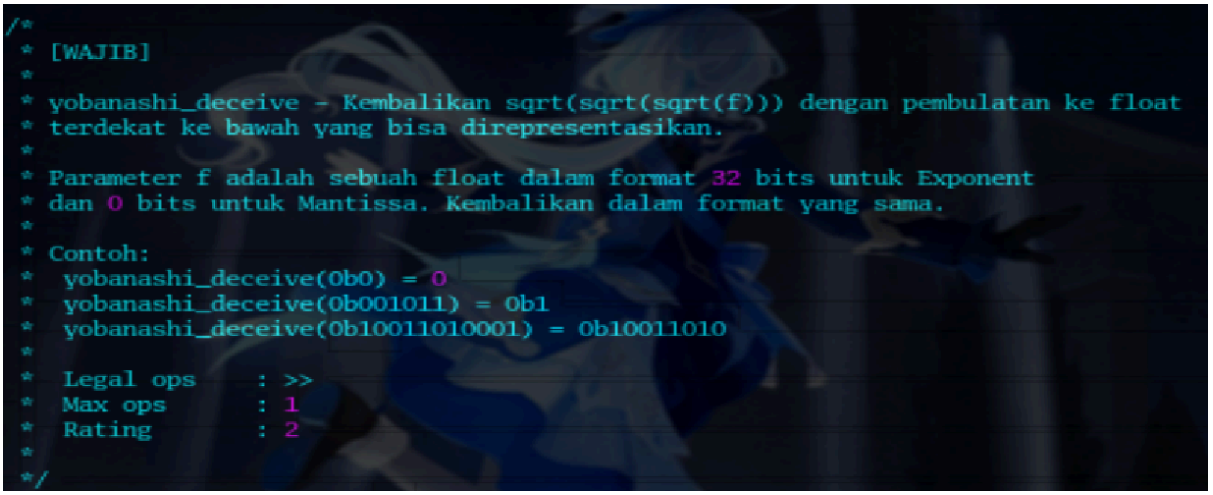
4 bit pertama sudah dihandle, sekarang saatnya untuk mengecek 4 bit kedua dari iofi. Sebelumnya, kita sudah melakukan shift ke kanan sebanyak 4 pada lofi, sehingga kita sudah punya nilai 4 bit keduanya. Nah karena kita ingin agar nilai 4 bit pertama 1111, maka nilai 4 bit keduanya haruslah 0000, sehingga kita lakukan operasi *not* (!) untuk mengecek apakah 4 bit keduanya bernilai 0000. Jika 4 bit keduanya 0000, maka isUp akan menghasilkan 1 (negasinya). Terakhir kita lakukan return dengan melakukan komparasi pada nilai 4 bit pertama dan 4 bit kedua, dan akhirnya soal ini AC.

Referensi

Tidak ada referensi.

Yobanashi Deceive

- Rating 2 [4 pts : 2 correct + 2 perfect]



```
/*
 * [WAJIB]
 *
 * yobanashi_deceive - Kembalikan sqrt(sqrt(sqrt(f))) dengan pembulatan ke float
 * terdekat ke bawah yang bisa direpresentasikan.
 *
 * Parameter f adalah sebuah float dalam format 32 bits untuk Exponent
 * dan 0 bits untuk Mantissa. Kembalikan dalam format yang sama.
 *
 * Contoh:
 *  yobanashi_deceive(0b0) = 0
 *  yobanashi_deceive(0b001011) = 0b1
 *  yobanashi_deceive(0b10011010001) = 0b10011010
 *
 *  Legal ops     : >>
 *  Max ops       : 1
 *  Rating        : 2
 *
 */
```

legal ops : >>
max ops : 1

Solving steps

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

Penjelasan

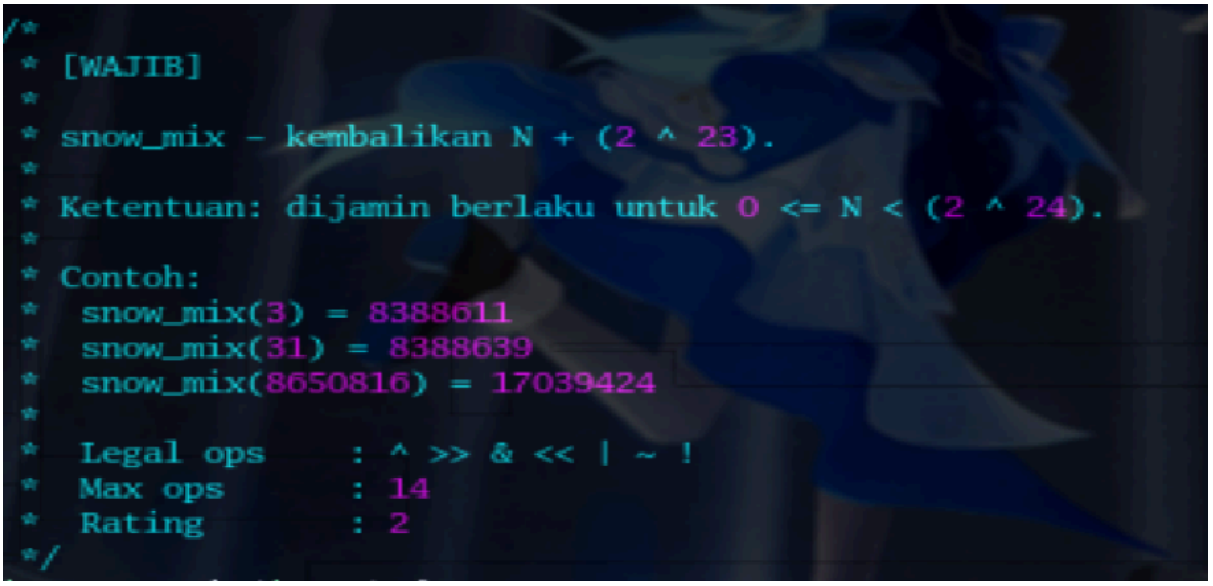
Nah, sekarang aku akan membahas tentang operasi shift kanan (>>). Sama seperti shift kiri, shift kanan juga dapat dijadikan sebagai operasi aritmatika. Jika shift kiri sama dengan “x * 2^m”, maka shift kanan menurunkan exponen secara logaritmik. Karena ada 3 sqrt, maka f di shift ke kanan sebanyak 3. Mengapa shift kanan bisa dipakai pada masalah ini? Karena ada constraint pembulatan float terdekat ke bawah. Sekian.

Referensi

https://stackoverflow.com/questions/13577174/can-anyone-explain-why-2-shift-means-divided-by-4-in-c-codes

Snow Mix

- Rating 2 [4 pts : 2 correct + 2 perfect]



```
/*
* [WAJIB]
*
* snow_mix - kembalikan N + (2 ^ 23).
*
* Ketentuan: dijamin berlaku untuk 0 <= N < (2 ^ 24).
*
* Contoh:
*   snow_mix(3) = 8388611
*   snow_mix(31) = 8388639
*   snow_mix(8650816) = 17039424
*
*   Legal ops     : ^ >> & << | ~ !
*   Max ops       : 14
*   Rating        : 2
*/
```

legal ops : ^, >>, &, <<, |, ~, !
max ops : 14

Solving steps

```
int snow_mix(int N) {
    int added = 1 << 23;
    int carry;

    carry = N & added;
    N = N ^ added;

    carry = carry << 1;
    added = N & carry;
    N = N ^ carry;

    carry = added << 1;
    N = N ^ carry;

    return N;
}
```

Penjelasan

Pada soal ini, kita diharuskan untuk *return* nilai N + (2 ^ 23). Pertama, aku membuat variabel untuk operasi 2 ^ 23, yakni “added = 1 << 23”, untuk alasan mengapa menggunakan left shift, sudah dijelaskan pada soal sebelumnya yang memakai metode ini juga (silahkan merujuk ke soal Chicken or Beef). Selanjutnya adalah implementasi variabel carry, variabel ini berguna untuk menyimpan nilai sementara. Kemudian, carry akan membawa nilai dengan bit ke 23 nya adalah 1 dan sisanya 0 (menyimpan representasi bit dari 2^23).

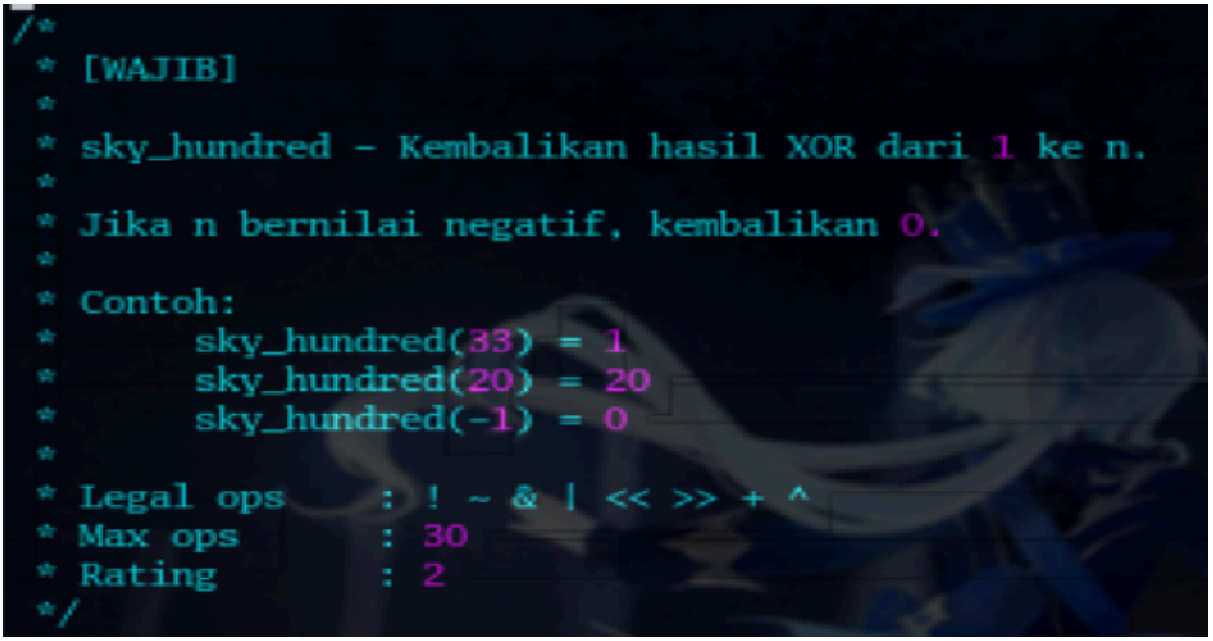
Sekarang, adalah algoritma untuk menambah 2^23 dengan N. Karena rangenya adalah 0 <= N < (2 ^24), maka saya akan mengisi sampai bit ke 25 (ini didapat secara *brute force* dengan mempertimbangkan constraint juga). Prosesnya adalah xor N dengan added untuk mengubah nilai, lalu nilai yang dibawa (carry) di geser ke kiri. Proses ini bertujuan untuk menampung hasil dari +N (menggeser ke kiri menjadi bit ke 24), lalu proses mirip dengan sebelumnya, lalu geser lagi ke kiri untuk mengisi bit ke 25, lakukan proses yang sama (xor). Ternyata, soal ini sudah cukup dengan handle 25 bit, sehingga didapatkan nilai dari N + (2^23).

Referensi

Tidak ada referensi.

Sky Hundred

- Rating 2 [4 pts : 2 correct + 2 perfect]



legal ops : !, ~, &, |, <<, >>, +, ^
max ops : 30

Solving steps

```
int sky_hundred(int n) {
    int neg = n >> 31;
    int mod4 = n & 3;

    int mod0 = !(mod4 ^ 0);
    int mod1 = !(mod4 ^ 1);
    int mod2 = !(mod4 ^ 2);
    int mod3 = !(mod4 ^ 3);

    int mask0 = mod0 << 31 >> 31;
    int mask1 = mod1 << 31 >> 31;
    int mask2 = mod2 << 31 >> 31;
    int mask3 = mod3 << 31 >> 31;

    int r0 = n & mask0;
    int r1 = 1 & mask1;
    int r2 = (n+1) & mask2;
    int r3 = 0 & mask3;
```

```
    int res = r0 | r1 | r2 | r3;

    return (neg & 0) | (~neg & res);
}
```

Penjelasan

Ada trik untuk mengerjakan soal ini, yakni dengan menggunakan mod4 (tolong merefer ke referensi). Diketahui bahwa jika n habis dibagi 4, maka program akan *return* n, jika sisa 1, *return* 1, jika sisa 2, *return* n+1, dan jika sisa 3, *return* 0.

Karena program juga mengecek kasus n negatif (dengan langsung *return* 0), aku mengecek kasus tersebut dalam variabel neg (n >> 31), neg merubah n menjadi -1 untuk negatif dan 0 untuk positif. Lalu, aku juga membuat variabel mod4 (n & 3) untuk menyimpan operasi dari n % 4, n % 4 dapat dituliskan juga sebagai n & 3, atau aturan umumnya adalah jika kita ingin n % m, maka dapat ditulis juga sebagai n & (m-1) (silahkan merujuk ke referensi).

Setelah menyimpan kasus negatif dan operasi dari mod4, saatnya mengecek hasil dari mod 4 tersebut, yang dijabarkan dengan variabel mod0 sampai mod3, arti dari “!(mod4 ^ hasil mod)” adalah sama dengan “mod4 == hasil mod” (silahkan merujuk ke referensi mengapa hal ini bisa terjadi), sehingga secara tak langsung kita mengecek setiap hasil dari mod4 dan menyimpannya pada variabel mod0 sampai mod3.

Setelah mendapatkan hasil dari mod4, cek apakah hasil mod itu benar untuk masing-masing kasus, jika salah, maka bitnya akan bernilai “0x00000000” atau 0, dan jika benar, maka bitnya akan bernilai “0x11111111”, atau bernilai -1. Pengecekan ini dilakukan pada variabel mask0 sampai mask3.

Setelah meng*handle* nilai mod, ekstrak hasil mod dengan *return* yang diinginkan dengan operator (&), ini terjadi pada variabel r0 sampai r3. Setelahnya, gabungkan semua kemungkinan hasil dengan operator (|), atau dengan kata lain, aku mengekstrak hasil dalam variabel res.

Terakhir, adalah return hasil. “`(neg & 0)`”, akan selalu menghasilkan 0, hal ini bertujuan untuk membandingkan apakah dia negatif atau tidak (sesuatu di-*or* akan menghasilkan sesuatu itu sendiri). Selanjutnya, dilakukan “`(~neg & res)`”, “`~neg`”

dilakukan untuk mengekstraksi bit dari hasil, jika positif maka bit yang awalnya semua 0 menjadi semua 1, dan sebaliknya. Terakhir, ekstrak hasil dari res, dan *return* hasilnya. Soal ini berhasil AC.

Referensi

Bagi yang penasaran :
- Ide :
https://stackoverflow.com/questions/33705785/how-to-take-the-xor-of-the-numbers-from-1-n-for-a-given-n-eg-123-n

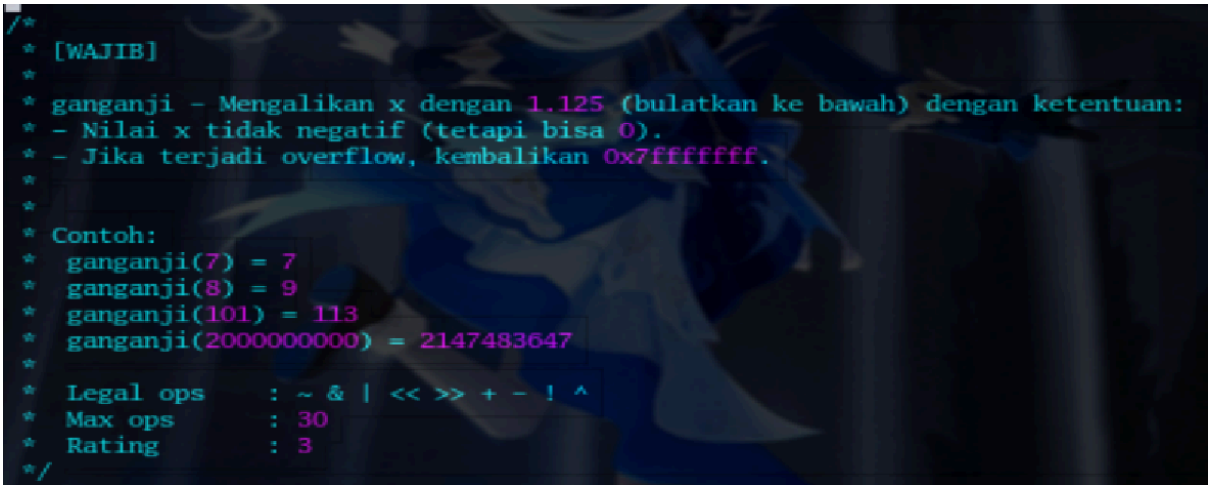
- mod operation with (&) :
https://stackoverflow.com/questions/3072665/bitwise-and-in-place-of-modulus-operator

- Equality (==) in bitwise :
https://stackoverflow.com/questions/4161656/replacing-with-bitwise-operators

- n << 31 >> 31 :
https://stackoverflow.com/questions/26192284/why-does-1-31-31-result-in-1

Ganganji

- Rating 3 [5 pts : 3 correct + 2 perfect]



legal ops : ~, &, |, <<, >>, +, -, !, ^
max ops : 30

Solving steps

```
int ganganji(int x) {
    int shift = x >> 3;
    int rets = x + shift;

    int ovf = rets >> 31;
    int maxi = ~(1 << 31);

    return (ovf & maxi) | (~ovf & rets);
}
```

Penjelasan

Pada soal ini, kita diharuskan untuk mengalikan x dengan 1.125 (atau sama dengan ⅛ jika diubah menjadi pecahan). Ketentuannya adalah, jika hasilnya tidak overflow, *return* hasil, sedangkan jika hasilnya overflow, *return* 0x7fffffff (atau sama dengan int max : 2^31-1).

Pertama-tama, saya membagi x dengan 8 dengan cara shift ke kanan 3 kali untuk membuat x menjadi lebih kecil dan mengurangi risiko terkena overflow. Selanjutnya, aku membuat variable rets untuk menampung hasil dari x dikali ⅛ .

Setelah mendapatkan hasil dari perkalian dengan 1.125, lakukan pengecekan apakah hasilnya overflow atau tidak (dalam hal ini, proses ini dilakukan pada variabel ovf). Pada variabel ovf, jika rets positif (tidak overflow), maka ovf akan bernilai 0 (0x00000000), sedangkan jika negatif (overflow), akan bernilai -1 (0x11111111).

Setelah dilakukan pengecekan, saatnya persiapan untuk melakukan *return*, yakni dalam hal ini aku membuat variabel maxi. Variabel maxi memiliki nilai int max (2^31-1).

Terakhir, adalah *return* hasil. Jika ovf tadi bernilai -1 (overflow), maka "`(ovf & maxi)`" akan bernilai maxi, karena ovf akan mengekstrak setiap bit dari max (sesuatu di *and* 1 akan mengambil bit yang bukan 0)i, dan nilai "`(~ovf & rets)`" akan bernilai 0 (0x00000000), karena sesuatu di *and* 0 akan menjadi 0. Nah dalam kasus overflow, *return* dapat disederhanakan menjadi "`(maxi | 0)`", yang pasti akan me-*return* 0x7fffffff (int max : 2^31-1).

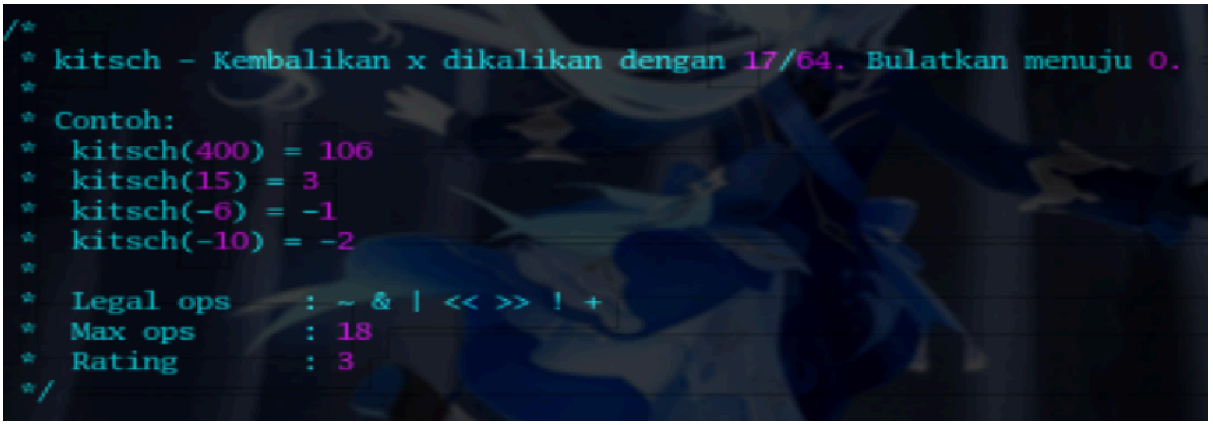
Untuk kasus tidak overflow, maka ovf akan bernilai 0 (0x00000000), menyebabkan nilai "`(ovf & maxi)`" akan bernilai 0, karena sesuatu di *and* 0 akan menjadi 0. Selanjutnya, "`(~ovf & rets)`" akan menghasilkan rets, karena sesuatu di *and* 1 akan mengambil bit yang bukan 0, sehingga *return* dapat disederhanakan menjadi "`(0 | rets)`", yang pasti akan me-*return* nilai dari rets.

Referensi

Tidak ada referensi.

Kitsch

- Rating 3 [5 pts : 3 correct + 2 perfect]



legal ops : ~, &, |, <<, >>, !, +
max ops : 18

Solving steps

```
int kitsch(int x) {
    int div = x >> 6;
    int m = (div << 4) + div;

    int r = x & 63;
    int round = ((r << 4) + r) >> 6;

    int pos = m + round;
    int neg = (x >> 31);

    return pos + (neg & (neg & !!r);
}
```

Penjelasan

Pada soal ini, kita diharuskan untuk mengalikan x dengan 17/64.. Ketentuannya adalah, *return* nilai x dikali 17/64 dengan melakukan pembulatan menuju 0 terlebih dahulu.

Pertama-tama, saya membagi x dengan 64 dengan cara shift ke kanan 6 kali untuk membuat x menjadi lebih kecil dan mengurangi risiko terkena overflow (nilai ditampung di variabel div). Selanjutnya, aku membuat variabel m untuk menampung

hasil dari x dari pembagian sebelumnya dikali dengan 17 (dikali 16 : “div << 4” +1 : “div”).

Setelah mendapatkan hasil dari perkalian dengan 17/64, buat skema pembulatannya, dalam hal ini, saya menyimpan nilai untuk pembulatan dalam round, sedangkan r adalah sama dengan “x mod 64” (karena dibagi 64, jadi ingin dilihat apakah ada sisa pembagian atau tidak). Setelah mendapat nilai round, pos ditambahkan dengan round untuk mendapatkan hasil yang sudah dibulatkan menuju nilai terkecil relatif dari positif (menuju 0, perhatikan sifat integer), hal ini dilakukan pada “`pos = m + round`”.

Setelah berhasil meng*handle* kasus positif, saatnya meng*handle* kasus negatif, karena ada perbedaan. Pada negatif, maka hasil dibulatkan menuju nilai terkecil relatif dari negatif (menjauhi 0, perhatikan sifat dari integer juga), tentunya kita harus menghandle ini. Nah, jika hasil mod 64 tadi sama dengan 0, maka hasil akan baik-baik saja, tetapi jika memiliki sisa, maka pembulatan akan menjauh dari 0, jadi harus kita *handle* terlebih dahulu. *Handle* negatif dimulai dari membuat pengecekan apakah bilangan tersebut negatif atau tidak (ada pada variabel neg).

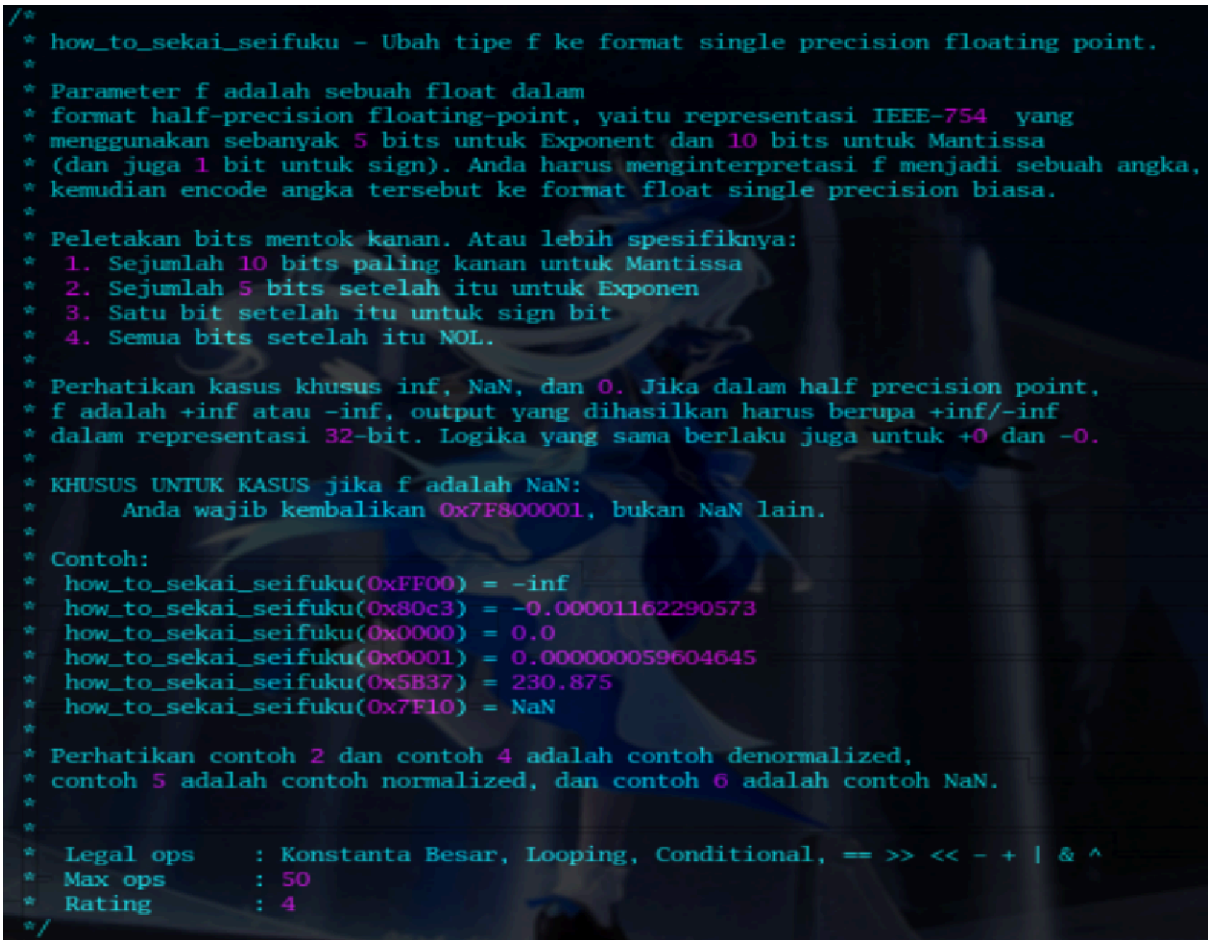
Terakhir, setelah mengecek apakah suatu hasil negatif atau tidak, sekarang aku akan membahas *return value*. Perhatikan variabel neg akan bernilai 0 (0x00000000) jika bernilai positif. Jika ini terjadi, maka ”`pos + (neg & (neg & !!r)`” akan bernilai pos. Sedangkan jika negatif, maka neg akan bernilai -1 (0x11111111), dan ”`(neg & (neg & !!r)`” akan bernilai 1 (0x00000001), pos + 1, maka hasil akan dibulatkan menuju 0 (anggap saja seperti -2 + 1 menjadi -1).

Referensi

Bagi yang mau baca :
https://stackoverflow.com/questions/12608159/multiply-by-5-8-and-watch-for-overflow

How To Sekai Seifuku

- Rating 4 [6 pts : 4 correct + 2 perfect]



```
/*
* how_to_sekai_seifuku - Ubah tipe f ke format single precision floating point.
*
* Parameter f adalah sebuah float dalam
* format half-precision floating-point, yaitu representasi IEEE-754  yang
* menggunakan sebanyak 5 bits untuk Exponent dan 10 bits untuk Mantissa
* (dan juga 1 bit untuk sign). Anda harus menginterpretasi f menjadi sebuah angka,
* kemudian encode angka tersebut ke format float single precision biasa.
*
* Peletakan bits mentok kanan. Atau lebih spesifiknya:
*  1. Sejumlah 10 bits paling kanan untuk Mantissa
*  2. Sejumlah 5 bits setelah itu untuk Exponen
*  3. Satu bit setelah itu untuk sign bit
*  4. Semua bits setelah itu NOL.
*
* Perhatikan kasus khusus inf, NaN, dan 0. Jika dalam half precision point,
* f adalah +inf atau -inf, output yang dihasilkan harus berupa +inf/-inf
* dalam representasi 32-bit. Logika yang sama berlaku juga untuk +0 dan -0.
*
* KHUSUS UNTUK KASUS jika f adalah NaN:
*      Anda wajib kembalikan 0x7F800001, bukan NaN lain.
*
* Contoh:
*  how_to_sekai_seifuku(0xFF00) = -inf
*  how_to_sekai_seifuku(0x80c3) = -0.00001162290573
*  how_to_sekai_seifuku(0x0000) = 0.0
*  how_to_sekai_seifuku(0x0001) = 0.0000000059604645
*  how_to_sekai_seifuku(0x5B37) = 230.875
*  how_to_sekai_seifuku(0x7F10) = NaN
*
* Perhatikan contoh 2 dan contoh 4 adalah contoh denormalized,
* contoh 5 adalah contoh normalized, dan contoh 6 adalah contoh NaN.
*
*
*  Legal ops     : Konstanta Besar, Looping, Conditional, == >> << - + | & ^
*  Max ops       : 50
*  Rating        : 4
*/
```

legal ops : Konstanta Besar, Looping, Conditional, ==, >>, <<, -, +, |, &, ^
max ops : 50

Solving steps

```
unsigned how_to_sekai_seifuku(unsigned f) {
    unsigned sign = (f >> 15) & 0x0001;
    unsigned exp = (f >> 10) & 0x001F;
    unsigned frac = f & 0x03FF;

    unsigned sign_s = sign << 31;
    unsigned exp_s, frac_s;
```

```
    if (exp == 0){
       if (frac == 0){
          return sign_s;
       }
       else{
           while ((frac & 0x0400) == 0){
               frac = frac << 1;
               exp = exp - 1;
           }
           exp = exp + 1;
           frac = frac & 0x03FF;
           exp_s = (127 - 15 + exp) << 23;
           frac_s = frac << 13;
           return sign_s | exp_s | frac_s
       }
    }
    else if (exp == 0x1f){
       if (frac == 0){
           return sign_s | 0x7F800000;
       }
       else{
           return 0x7F800000;
       }
    }

    exp_s = (exp + (127 - 15)) << 23;
    frac_s = frac << 13;

    return sign_s | exp_s | frac_s;
}
```

Penjelasan

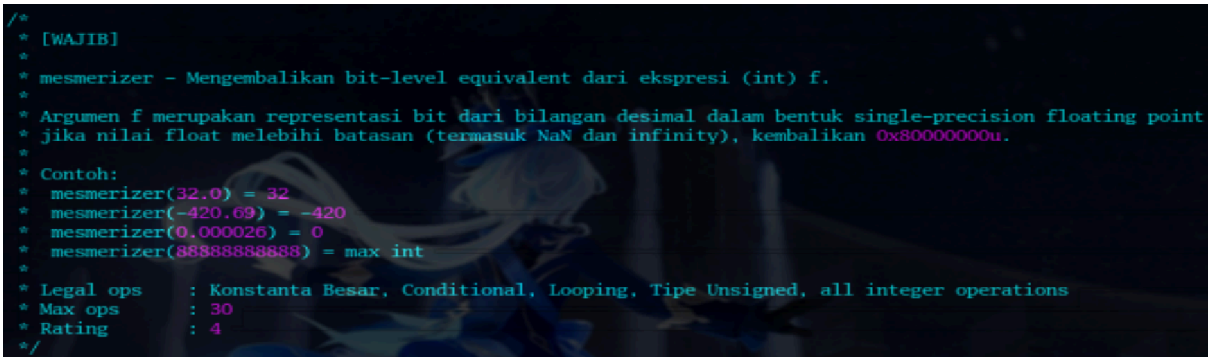
Soal ini tidak dibuat *Write-Up*nya (konversi dari soal ini : Shikanoko Nokonoko).

Referensi

Bagi yang mau baca :
https://www.fox-toolkit.org/ftp/fasthalffloatconversion.pdf

Mesmerizer

- Rating 4 [6 pts : 4 correct + 2 perfect]

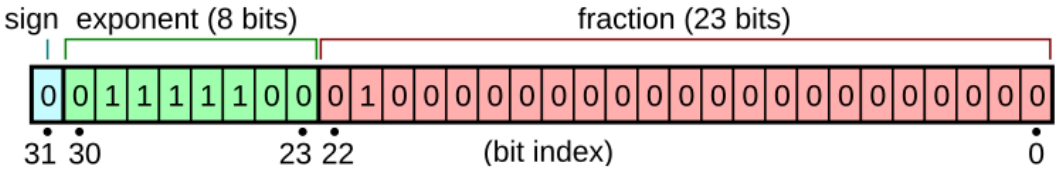


legal ops : Konstanta Besar, Conditional, Looping, Tipe Unsigned, all integer operations
max ops : 30

Solving steps

```
int mesmerizer(unsigned uf) {
    unsigned sign = uf >> 31;
    int exp = (uf >> 23) & 0xFF;
    int mantissa = (uf & 0x7FFFFF) | 0x800000;
    int nexp = exp - 127;
    int bin;

    if (exp == 0xFF){
       return 0x80000000u;
    } if (nexp < 0){
       return 0;
    } else if (nexp <= 23){
       bin = mantissa >> (23 - nexp);
    } else if (nexp <= 31){
       bin = mantissa  << (nexp - 23);
    } else{
       if (sign){
         return 0x80000000u;
       } else{
         return 0x80000000u;
       }
    }
    if (sign){
        bin = -bin;
    }
    return bin;
}
```

Penjelasan



Gambar di atas merupakan gambaran dari float IEEE-754. Dibagi menjadi 3 bagian, yakni sign, exponent, dan fraction (mantissa). Pada kode yang saya lampirkan, saya mengambil nilai sign di variabel sign, exponent di variabel exp, dan fraction (mantissa) di variabel mantissa.

Selanjutnya adalah mengecek nilai dari float yang dimasukkan, jika exponentnya bernilai 0xFF, maka exponent tersebut antara Not a Number (NaN) atau inf dan -inf. Jika benar bernilai 0xFF, maka program akan me*return* 0x80000000u sesuai yang diminta pada spesifikasi.

Selanjutnya adalah pengecekan nilai nexp, sebelumnya apa itu nexp? Di sini, nexp adalah nilai exponent yang sudah dinormalisasi sesuai format IEEE-754 (absolute exponent), nah jika nilai nexp ini bernilai kurang dari 0 (< 0), maka nilai exponentnya negatif, menyebabkan nilai float pasti kurang dari 1 (contoh 2^-3 = ⅛ = 0.125). Karena kita ingin me*return* bit-level equivalentnya, maka kita pasti akan me*return* 0 (pembulatan menuju 0).

Selanjutnya adalah pengecekan nilai nexp lagi, tapi relatif terhadap nilai mantissa. Jika nexp <= 23, maka mantissa akan bergeser ke kanan sebanyak “23 - nexp”. Mengapa hal ini terjadi? Karena nexp menunjukkan desimal di belakang float tidak terlalu besar, sehingga kita bulatkan ke bawah (shift ke kanan = bagi) Contoh, 1.4 menjadi 1. Sedangkan jika nexp bernilai “23 < nexp <= 31”, maka mantissa akan bergeser ke kiri sebanyak (nexp - 23). Mengapa hal ini terjadi? karena nexp menunjukkan desimal di belakang float cukup besar, sehingga kita bulatkan ke atas (shift ke kanan = kali). Contoh, 0.6 menjadi 1. Perhatikan bahwa pengurangan nexp terhadap 23 bergantung dari siapa yang lebih besar (menghindari hasil negatif). Mengapa dikurangi 23? Karena mantissa berukuran 23 bits.

Selanjutnya adalah pengecekan overflow float dari sign, jika benar, maka program akan me*return* 0x80000000u. Terakhir, adalah mengecek apakah float negatif atau positif dengan cara melihat nilai dari sign. Jika sign bernilai 1, maka float bernilai negatif, jadi bit-level equivalentnya dikali dengan -1 agar menjadi negatif, dan taraa~~~, tinggal *return* bit-level equivalentnya aja (variabel bin).

Referensi

Bagi yang mau baca :
https://en.wikipedia.org/wiki/IEEE_754

https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/

Akhir Kata

Sekian, Write Up dari aku, Katou, semoga AC, dan semoga 100 😭!

(*inhales, bismillah Hology*)

Ehe ~~~ :3

“Jarvis, nyalakan Chat GPT”

-> Katou 😜