WRITE-UP

Nama: Zulfaqqar Nayaka Athadiansyah
NIM: 13523094
KELAS: K02

Soal 1 – chicken_or_beef

1. Solusi

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

2. Penjelasan Singkat

Pada soal ini, kita diminta menerapkan operasi bitwise OR pada 4 bit paling kanan dari `beef*2` dan 4 bit paling kiri dari `chicken`.

Teknik yang mendasari solusi ini adalah *bit masking*, yakni teknik menyimpan bit yang diinginkan dan membuang (dengan kata lain, membuatnya menjadi 0) bit yang tidak diinginkan. Di sini, masking diterapkan untuk mendapatkan 4 bit saja dan mematikan bit sisanya.

`beef*2` dihitung dengan melakukan shift ke kiri sebanyak satu kali. Selanjutnya, diterapkan *bit masking* pada 4 bit pertamanya dengan menerapkan operasi AND dengan 15 (0b00001111) padanya.

Sementara itu, 4 bit paling kiri pada `chicken` didapatkan dengan melakukan shift ke kanan sebanyak 4 kali. Selanjutnya, sama seperti sebelumnya, kita terapkan *bit masking* untuk mendapatkan 4 bit paling kanan saja.

Sisanya adalah mengembalikan bitwise OR dari kedua nilai yang kita dapatkan.

3. Referensi yang Digunakan

- Tidak ada.

Soal 2 – masquerade

1. Solusi

```
int masquerade() {
    return (128 << 24) ^ 1;
}
```

2. Penjelasan Singkat

Pada soal ini, kita diminta untuk menentukan bilangan terkecil kedua dalam representasi *integer two's complement*. Bilangan terkecilnya adalah `10000000000000000000000000000000` sehingga bilangan terkecil kedua adalah `10000000000000000000000000000001`, yang bisa didapatkan dengan operasi *bitwise* XOR:

`10000000000000000000000000000000 ^ 00000000000000000000000000000001`

Bilangan pertama didapatkan dengan `(1 << 31)` atau `(128 << 24)`.

3. Referensi yang Digunakan

- Tidak ada.

Soal 3 – airani_iofifteen

1. Solusi

```c
int airani_iofifteen(int iofi) {
    int masked = iofi & 15;
    int digit3 = (masked >> 3) & 1;
    int digit2 = (masked >> 2) & 1;
    int digit1 = (masked >> 1) & 1;
    int digit0 = masked & 1;
    return !(iofi >> 4) & digit0 & digit1 & digit2 & digit3;
}
```

2. Penjelasan Singkat

Pada soal ini, kita diminta untuk menentukan nilai dari `iofi == 15`. Jika sama, keluarkan `1`; jika tidak, `0`.

Pertama-tama, kita akan mengambil tiap digit dari 4 bit paling kanan untuk menentukan kesamaannya dengan `15`. `iofi` di-*masking* dahulu untuk mengambil 4 bit pertamanya (`iofi & 15 == iofi & 0b00001111`). Selanjutnya, kita menerapkan shift dan *masking* dengan `1` untuk mendapatkan tiap digit dari `iofi`.

Untuk memeriksa kesamaan, triknya di sini adalah memanfaatkan fakta bahwa `15 == 0x00001111`. Perhatikan 4 bit paling kanan. Jika kita terapkan operasi *bitwise* AND pada seluruh bit tersebut maka akan didapatkan `1`. Akibatnya, untuk memeriksa apakah suatu bilangan 4 bit sama dengan `15`, kita cukup AND-kan seluruh digitnya saja. Jika sama maka hasilnya adalah `1`. Selebihnya, kita akan dapatkan `0`.

Tentunya jika metodenya hanya sebatas itu, bilangan yang memiliki `1111` di paling kanan juga akan bernilai benar, misalnya `iofi = 31 = 0b00011111` ataupun `iofi = 79 = 0b00101111` akan menghasilkan `1` karena kita lupa memperhitungkan bit-bit selain 4 bit paling kanan. Dengan kata lain, kita harus tentukan apakah seluruh bit selain 4 bit paling kanan bernilai `0` atau tidak.

Solusi dari masalah ini adalah memanfaatkan operator logika NOT (`!`). Nilai dari `!(x)` adalah `0` jika `x` adalah bilangan selain `0` dan `1` jika `x` adalah `0`. Jadi, untuk kasus ini kita bisa menerapkan operator logika NOT pada bit setelah 4 bit paling kanan (didapatkan dengan `iofi >> 4`). Terakhir, kita cukup *bitwise* AND-kan seluruh 4 digit dan `!iofi >> 4`.

3. Referensi yang Digunakan

Tidak ada.

Soal 4 – yobanashi_deceive

1. Solusi

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

2. Penjelasan Singkat

Diberikan `f` berupa float dengan 32 bits untuk *exponent* dan 0 bits untuk *mantissa*, soal meminta kita untuk memberikan `sqrt(sqrt(sqrt(f)))`. Untuk orang yang kurang fokus (saya), sekilas soal ini awalnya memang men-'deceive' :D

Mengingat formatnya adalah 32 bits untuk *Exponent*, idenya adalah sesimpel mengetahui bahwa

$$\sqrt{\sqrt{\sqrt{x^k}}} = x^{k/8}$$

dan

$$\texttt{a >> x} = a \times 2^x.$$

Untuk mendapatkan hasil yang diinginkan, kita cukup membagi *exponent*-nya dengan $2^3 = 8$, yang bisa didapatkan dengan melakukan shift `3` kali ke kanan.

3. Referensi yang Digunakan

IEEE Standard for Floating-Point Arithmetic (IEEE 754)

Soal 5 – snow_mix

1. Solusi

```
int snow_mix(int N) {
    int x = 1 << 23;
    return ((N&x)<<1)^(N^x);
}
```

2. Penjelasan Singkat

Pada soal ini, kita akan menjumlahkan $x = 2^{23}$ dengan N $(0 \leq N < 2^{24})$. Idenya bisa didapatkan melalui *trial-and-error* untuk melakukan penjumlahan dua bilangan dalam representasi biner:

```
\\ nilai awal
   x                                  =   01000000000000000000000000
   y                                  =   01000010000000000001000000

\\ carry
   x&y                                =   01000000000000000000000000
   (x&y)<<1                           =   10000000000000000000000000

\\ sum tanpa carry
   x^y                                 =   00000010000000000001000000

\\ sum dengan carry
   (x^y)^((x&y)<<1)                    =   10000010000000000001000000

\\ (supposedly) carry selanjutnya
  (x^y)&((x&y)<<1)                     =   00000000000000000000000000

\\ dan seterusnya...
```

Kita bisa *meminjam*[1] ide "pinjam" dari penjumlahan bersusun yang dikenal saat TK/SD. Untuk basis-n, ketika hasil penjumlahan dari suatu digit melebihi $n - 1$ maka kita cukup ambil digit satuannya saja dan 1 akan "dipinjamkan" ke digit yang lebih besar (ke kiri). Dalam bahasa Inggris, konsep "pinjam-meminjam" ini disebut *carry*. Penjumlahan dua bilangan biner 1 bit hanya memiliki 3 kemungkinan untuk 4 kondisi:

$$0 + 0 = 0;\ 1 + 0 = 0 + 1 = 1;\ 1 + 1 = 10$$

[1] *Pun unintended.*

Pada kasus $1 + 1 = 10$ 'lah konsep *carry* digunakan. Kita bisa mengetahui digit mana saja yang mengalami kasus ini dengan menerapkan operasi bitwise AND (`x&y`) karena $1\&1 = 1$, selebihnya 0. Untuk menjumlahkan *carry* ke digit selanjutnya dilakukan shift sekali ke kiri (`x & y << 1`).
Sementara itu, untuk mendapatkan hasil tanpa *carry*, kita bisa gunakan operasi bitwise XOR (`x^y`) pada dua bilangan yang ingin dijumlahkan, pada kasus ini 2^{23} dan N. Hal ini dikarenakan XOR dari dua bilangan biner 1 bit mempunyai 2 kemungkinan untuk 4 kondisi:

$$0 \oplus 0 = 1 \oplus 1 = 0; \quad 1 \oplus 0 = 0 \oplus 1 = 1$$

Di sini, hasil yang paling bermanfaat adalah $1 \oplus 1 = 0$ karena menggambarkan hasil penjumlahan pada digit tersebut tanpa memperhitungkan *carry*. Untuk mendapatkan hasil penjumlahan dengan *carry*, terapkan *bitwise* XOR pada hasil penjumlahan tanpa *carry* (`x^y`) dengan *carry* (`x&y<<1`).

Untuk kasus penjumlahan dua bilangan secara umum, proses ini akan diiterasi karena penjumlahan barusan bisa jadi menghasilkan *carry* yang baru. Iterasi ini akan dihentikan ketika *carry* bernilai nol.

Akan tetapi, mengingat pada kasus ini $x = 2^{23} =$ `01000000000000000000000` dan $0 \leq N < 2^{24}$, hanya akan terjadi paling banyak 1 kali *carry*, misalnya pada kasus $N = 2^{23} + 1$:

```
\\ nilai awal
   x                                  =   01000000000000000000000
   N                                  =   01000000000000000000001
\\                                          ^
\\                                 Di sini akan terjadi carry
\\ Hasilnya...
    x + N = (x^N)^((x&N)<<1)           =   10000000000000000000001
\\ Tidak ada carry lagi!
```

3. Referensi yang Digunakan

Tidak ada.

Soal 6 – sky_hundred

1. Solusi

```c
int sky_hundred(int n) {
    int rem = n & 3;
    int neg = ~(n >> 31);
    int mod0 = n & (~(!rem) + 1) & neg;
    int mod1 = 1 & (~(!(rem ^ 1)) + 1) & neg;
    int mod2 = (n + 1) & (~(!(rem ^ 2)) + 1) & neg;
    return (mod0 + mod1 + mod2);
}
```

2. Penjelasan Singkat

Pada soal ini, kita diminta untuk menentukan hasil dari

$$f(n) = 1 \oplus 2 \oplus 3 \oplus \ldots \oplus n.$$

Idenya bisa didapatkan dengan mengamati pola nilai $f(n)$.

- $f(n) = n$ jika $n \bmod 4 = 0$;
- $f(n) = 1$ jika $n \bmod 4 = 1$;
- $f(n) = n + 1$ jika $n \bmod 4 = 2$; dan
- $f(n) = 0$ jika $n \bmod 4 = 3$.

$n \bmod 4$ didapatkan dengan me-*masking* 2 bit pertama (`n & 3`) karena k bit paling kanan merepresentasikan nilai $n \bmod 2^k$. Misalnya, untuk $k = 2$:

$$n = 0 = 00 \rightarrow n \bmod 4 = 0 = 00 = n$$

$$n = 1 = 01 \rightarrow n \bmod 4 = 1 = 01 = n$$

$$n = 2 = 10 \rightarrow n \bmod 4 = 2 = 10 = n$$

$$n = 3 = 11 \rightarrow n \bmod 4 = 3 = 11 = n$$

Selanjutnya, kita bagi menjadi tiga kasus:
Kasus $n \bmod 4 = 0$ dilambangkan dengan variabel `mod0`.
Kasus $n \bmod 4 = 1$ dilambangkan dengan variabel `mod1`.
Kasus $n \bmod 4 = 2$ dilambangkan dengan variabel `mod2`.
Pada baris 4–6, kegunaan ekspresi seperti `rem ^ 1` dan `rem ^ 2` adalah menguji kesamaan dari `rem` dengan nilai yang bersesuaian (1 atau 2). Ketiga baris tersebut

pada dasarnya melakukan masking terhadap `n`, `1`, dan `n+1` berdasarkan kondisi yang terjadi, yakni nilai dari $n \, mod \, 4$ dan apakah bilangannya negatif.

Untuk menangani kasus negatif, kita menggunakan *mask* `int neg = ~(n >> 31)`. *Mask* ini menghasilkan nilai `00000000000000000000000000` jika n negatif dan `11111111111111111111111111` jika n nonnegatif. Ini penting untuk memastikan bahwa hanya nilai non-negatif yang diperhitungkan dalam hasil akhir. Kita akan terapkan *bitwise* AND antara *mask* ini pada tiap kasus modulo untuk menghasilkan 0 jika n negatif, dan menghasilkan jawaban sesungguhnya jika n positif (karena di-*masking* oleh `11111111111111111111111111`).

Untuk kasus $n \, mod \, 4 = 3$, nilai dari variabel `mod0`, `mod1`, `mod2` adalah 0 sehingga ketika dijumlahkan juga 0. Akibatnya, tidak perlu membuat variabel khusus `mod3`, cukup jumlahkan saja nilai variabel ketiga kasus lainnya.

3. Referensi yang Digunakan

https://www.geeksforgeeks.org/calculate-xor-1-n/

Soal 7 – ganganji

1. Solusi

```
int ganganji(int x) {
    int res = x + (x >> 3);
    int overflow = (res >> 31) ^ (x >> 31);
    int nmax = ~(1 << 31);
    res = ~(overflow) & res;
    return (res + (overflow & nmax));
}
```

2. Penjelasan Singkat

Pada soal ini, kita diminta untuk menghitung $1.125\mathrm{x}$, atau $\frac{9}{8}x$. Andai tidak perlu mempertimbangkan kasus *overflow*, program ini sebenarnya bisa sebatas `return x + (x >> 3);`.

Kasus *overflow* terjadi ketika tanda dari `x` tidak sama dengan tanda dari `x + (x >> 3)`. Hal ini ditangani pada variabel `overflow` yang akan bernilai `00000000000000000000000000` jika **tidak** terjadi *overflow* dan `11111111111111111111111111` jika terjadi *overflow*. Variabel ini nantinya dipakai untuk melakukan *masking* pada `res` dan `nmax` $= 2^{31} - 1$.

Ketika terjadi *overflow*, `~(overflow)=0` sehingga `~(overflow)&res=0`. Adap Pada kasus sebaliknya, operasi tersebut menghasilkan `res`. Jadi, fungsi dari proses tersebut adalah “mematikan” `res` ketika terjadi *overflow*. Selanjutnya, nilai dari `overflow & nmax` adalah `nmax` jika terjadi *overflow* dan `0` sebaliknya. Dengan demikian, mekanisme ini memungkinkan kita untuk memberikan keluaran tergantung kondisi *overflow* tanpa menggunakan percabangan (*if-else*).

3. Referensi yang Digunakan

Tidak ada.

Soal 8 – kitsch

1. Solusi

```
int kitsch(int x) {
    int xper64 = x >> 6; // floor(x/64)
    int rem64 = x & 63; // x mod 64
    xper64 += xper64 << 4; // 17 * floor(x/64)
    rem64 += rem64 << 4; // 17 * (x mod 64)
    rem64 += x >> 31 & 63; // rem64++ kalau x negatif
    xper64 += rem64 >> 6; // 17 * floor(x/64) + 17/64 * (x mod 64)
    return xper64;
}
```

2. Penjelasan Singkat

Pada soal ini, kita diminta menghitung $\frac{17}{64}x$ dibulatkan menuju nol. Bagaimana idenya? Misalkan $[x] = floor(x)$. Dengan ini, suatu bilangan x dapat dinyatakan sebagai

$$x = n\left[\frac{x}{n}\right] + x \bmod n$$

untuk $n \in \mathbb{N}$ sehingga untuk $n = 64$ diperoleh

$$x = 64\left[\frac{x}{64}\right] + x \bmod 64.$$

Mengalikan kedua ruas dengan 17 menghasilkan

$$\frac{17}{64}x = 17\left[\frac{x}{64}\right] + \frac{17}{64}(x \bmod 64).$$

Dengan persamaan di atas, kode solusi menjadi cukup *straightforward* dengan mencatat bahwa $x \bmod 2^y$ bisa didapatkan dengan $x \,\&\, (2^y - 1)$ untuk menerapkan *masking* pada y digit pertama. Selain itu, $\left[\frac{x}{2^y}\right] = x >> y$.

Kasus $x < 0$ ditanggulangi oleh baris ke-6 (`rem64 += x >> 31 & 63`), di mana $17x \bmod 64$ di-*increment* supaya dibulatkan menuju 0.

3. Referensi yang Digunakan

Tidak ada. Semuanya datang dari mimpi.

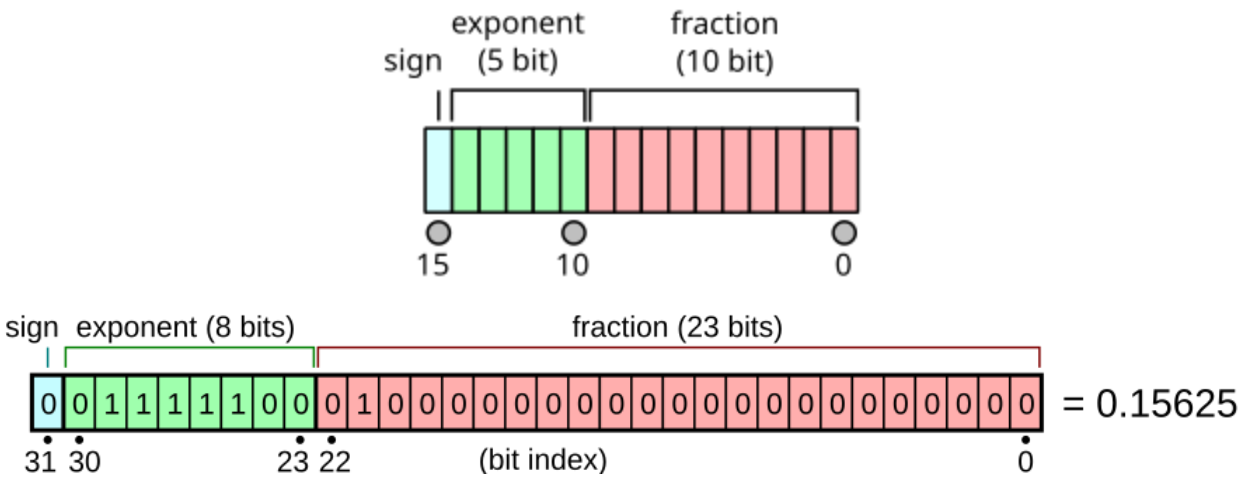
Soal 9 – how_to_sekai_seifuku

1. Solusi

```
unsigned how_to_sekai_seifuku(unsigned f) {
    unsigned sign = (f >> 15) & 1;
    unsigned exp = (f >> 10) & 31; // masking 5 bit exp dengan & 11111
    unsigned mantissa = f & 1023; // masking 10 bit mantissa dengan
1111111111
    unsigned res = sign << 31;
    // NaN dan +- Inf
    if (exp == 31) {
        if ((mantissa & 1023) == 0) return res | 0x7F800000; // +- inf
        else return 0x7F800001; // NaN
    }
    if (exp == 0) {
        if (mantissa == 0) return res; // Nol bertanda
        while ((mantissa & (1 << 10)) == 0)
            exp -= 1;
            mantissa <<=  1;
        }
        exp += 1;
        mantissa &= 1023;
    }
    exp += 112;
    return res | (exp << 23) | (mantissa << 13);
}
```

2. Penjelasan Singkat

Pada soal ini, kita diminta mengkonversi dari *half-precision floating point* menjadi *single-precision floating-point*. Dari segi representasi bit, keduanya terdiri atas *sign*, *exponent*, dan *mantissa* dengan rincian pembagian bit sebagai berikut:



Untuk mendapatkan *sign* kita cukup melakukan shift ke kanan lalu me-*masking*-nya dengan 1. Untuk *exponent*, kita lakukan shift 10 ke kanan lalu di-*masking* dengan 31 = 11111. Adapun *mantissa* dapat diekstrak dengan menerapkan bitwise AND padanya dengan 1023 = 1111111111 karena tersusun atas 10 bit.

Selanjutnya, kita atasi kasus ∞, $-\infty$, atau `NaN`. Hal ini terjadi ketika seluruh bit pada *exponent* bernilai 1, atau dengan kata lain, *exponent* sama dengan 11111= 31. Jika *mantissa*-nya sepenuhnya bernilai 0 maka `f` bernilai ∞, $-\infty$ (tergantung nilai *sign*). Selebihnya (ada bit bernilai 1 pada *mantissa*), maka `f` bernilai `NaN`.

Setelah itu, kita gunakan *while-loop* untuk melakukan normalisasi pada *mantissa* semula ke *single-precision*. Terakhir, kita juga harus mengubah biasnya. Bias bernilai 128 untuk *single precision* dan 15 untuk half precision. Akibatnya, kita jumlahkan *exponent* dengan 127 -15 = 112. Langkah yang tersisa adalah menggabungkan *sign*, *exponent*, dan *mantissa* dengan bitwise OR.

3. Referensi yang Digunakan

Jeroen van der Zijp. Fast Half Float Conversions.
http://www.fox-toolkit.org/ftp/fasthalffloatconversion.pdf
https://cs.stackexchange.com/questions/130384/how-are-bitwise-operators-used-in-normalisation-of-floating-point-numbers
https://en.wikipedia.org/wiki/Single-precision_floating-point_format
https://en.wikipedia.org/wiki/Half-precision_floating-point_format
https://evanw.github.io/float-toy/

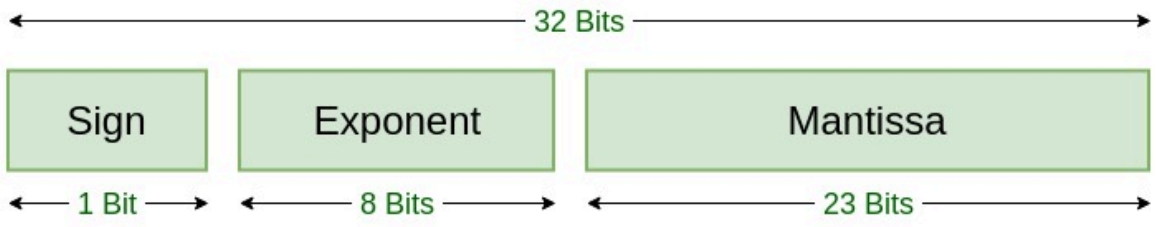
Soal 10 – mesmerizer

1. Solusi

```c
int mesmerizer(unsigned uf) {
    unsigned max_int = 1 << 31;
    unsigned sign = uf >> 31;
    unsigned exp = ((uf >> 23) & 255);
    unsigned frac = uf & ((1 << 23) - 1);
    if (exp == 255) return max_int; // kasus inf, -inf, atau NaN
    exp = exp - 127; // mengurangi dengan bias
    if (exp >> 31 & 1) return 0; // pecahan
    if (!(((exp - 30) >> 31) & 1)) return max_int; // exp > 30, overflow
    int x = 1 << exp; // menghitung 2^exp
    if (((exp - 23) >> 31) & 1) {
        x |= frac >> (23 - exp);
    } else {
        x |= frac << (exp - 23);
    }
    if (sign == 1) x = ~x + 1;
    return x;
}
```

2. Penjelasan Singkat

Pada soal ini, kita akan meng-*casting* float `uf` menjadi integer. Di sini kita akan mengacu pada standar IEEE 754:



Single Precision
IEEE 754 Floating-Point Standard

Dengan operasi *shift* dan teknik *masking*, kita dapatkan tanda (*sign*, bit paling kiri), bagian eksponen (bit ke 24 hingga bit ke 31), dan bagian *mantissa*-nya (23 bit

pertama). Sebelum memproses lebih lanjut, kita bisa *rule out* kemungkinan jika eksponennya adalah 255 = 11111111, yakni ketika `uf` bernilai ∞, − ∞, atau `NaN`. Pada kasus ini, kita akan mengeluarkan `0x80000000u`.

Selanjutnya, catat bahwa `E = exp - Bias` dengan `Bias = 127` untuk *single precision*. Nilai E adalah eksponen sesungguhnya dari 2. Jika eksponen negatif (artinya nilai terlalu kecil), fungsi akan mengembalikan 0. Jika eksponen lebih besar dari 30, hasilnya akan terlalu besar, sehingga fungsi mengembalikan nilai maksimum integer.

Fungsi kemudian menghitung 2^E dan kemudian menyesuaikan mantissa berdasarkan posisi eksponen. Jika bit tanda menunjukkan negatif, hasilnya diubah menjadi bentuk negatif. Terakhir, fungsi mengembalikan nilai integer yang sesuai dengan representasi floating-point `uf`.

3. Referensi yang Digunakan

IEEE Standard for Floating-Point Arithmetic (IEEE 754)
https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/
https://evanw.github.io/float-toy/