WRITE-UP

Nama: Daniel Pedrosa Wu
NIM: 13523099
Kelas: K01

Disclaimer: Saya merujuk pada bits menggunakan zero-indexing.

Soal 1 – chicken_or_beef

1. Solusi

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

2. Penjelasan
   a. Soal ini meminta praktikan untuk mengekstrak 4 bits kedua (bit ke-4 hingga bit ke-7) dari integer chicken dan menggunakan operasi bitwise OR untuk mengekstrak 4 bits pertama (bit ke-0 hingga bit ke-3) dari integer beef dikali 2 (beef * 2).
   b. Untuk mengekstrak 4 bits kedua dari integer chicken, chicken digeser ke kanan sebanyak 4 bits. Untuk mematikan bit lainnya, lakukan operasi bitwise AND antara chicken >> 4 dan 15. Dalam representasi biner, 15 adalah 00000000 00000000 00000000 00001111 sehingga yang tersisa hanya 4 bits pertama dari chicken >> 4.
   c. Untuk mengekstrak 4 bits pertama dari integer (beef * 2), kita perlu mencari beef * 2. Karena operator * tidak diperbolehkan, kita harus menggunakan operasi bitwise left shift sebanyak 1 bit. Ingat bahwa:
   $$a \ll n = a \times 2^n$$
   Bit lainnya dapat dimatikan dengan melakukan operasi bitwise AND antara beef << 1 dengan 15.
   d. Selanjutnya keduanya digabung dengan operasi bitwise OR
3. Referensi
   Tidak ada

Soal 2 – masquerade

1. Solusi

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

2. Penjelasan
   a. Soal ini meminta praktikan untuk mengembalikan angka terkecil kedua dalam representasi signed integer two's complement tanpa menggunakan konstanta besar.
   b. Dalam representasi signed integer two's complement, nilai integer terbesar adalah $2^{31}-1$, sementara nilai integer terkecil adalah -2^{31}. Artinya, nilai integer terkecil kedua adalah $-2^{31}+1=-(2^{31}-1)$ atau bentuk negatif dari nilai integer terbesar. Representasi bit dari nilai integer terbesar adalah 01111111 11111111 11111111 11111111. Bentuk negatifnya didapat dengan melakukan operasi bitwise NOT lalu ditambah dengan 1.
   c. 01111111 11111111 11111111 11111111 dioperasikan dengan bitwise NOT.

$$\sim \frac{01111111\ 11111111\ 11111111\ 11111111}{10000000\ 00000000\ 00000000\ 00000000}$$

   Setelah itu, tambahkan 1 ke representasi biner tersebut. Karena tidak diperbolehkan menggunakan operator +, maka kita operasikan dengan operator XOR dengan 1 (00000000 00000000 00000000 00000001). Dengan ini, bit terakhirnya akan menjadi 1 dan tandanya juga 1 (bit ke-31 dalam two's complement bernilai 1 menandakan bilangan negatif).

$$\sim \frac{\begin{array}{l}10000000\ 00000000\ 00000000\ 00000000\\ \underline{00000000\ 00000000\ 00000000\ 00000001}\end{array}}{10000000\ 00000000\ 00000000\ 00000001}$$

3. Referensi
   Tidak ada

Soal 3 – airani_iofifteen

1. Solusi

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

2. Penjelasan
   a. Soal ini meminta praktikan untuk mengembalikan 1 jika nilai integer iofi adalah 15, selain itu mengembalikan 0.
   b. Bilangan integer 15 memiliki representasi biner 00000000 00000000 00000000 00001111. Pertama, kita perlu memeriksa apakah semua bit di atas bit ke-3 bernilai 0. Hal ini dilakukan dengan menggesernya ke kanan sebanyak 4 bits (iofi >> 4). Kita gunakan operasi logical NOT untuk mengevaluasi biner dari (iofi >> 4). Logical NOT mengembalikan 1 jika binernya bernilai 0 sehingga kita mengembalikan 1 jika semua bit di atas bit ke-3 bernilai 0.
   c. Selanjutnya, kita periksa satu per satu bit yang berada di 4 bits pertama. Ini dilakukan dengan menggesernya ke kanan dan menggunakan operator bitwise AND untuk mengecek apakah nilai bit tersebut bernilai 1 atau tidak.
      i. (iofi & 1) mengecek apakah bit ke-0 bernilai 1
      ii. (iofi >> 1) & 1 mengecek apakah bit ke-1 bernilai 1
      iii. (iofi >> 2) & 1 mengecek apakah bit ke-2 bernilai 1
      iv. (iofi >> 3) & 1 mengecek apakah bit ke-3 bernilai 1
   d. Setelah semua pengecekan tersebut, kita gabungkan semuanya dengan operator bitwise AND. Fungsi akan mengembalikan 1 jika dan hanya jika bit ke-0 hingga bit ke-3 bernilai 1 dan sisanya bernilai 0.

3. Referensi
   Tidak ada

Soal 4 – yobanashi_deceive

1. Solusi

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

2. Penjelasan
   a. Soal ini meminta praktikan untuk mengembalikan nilai $\sqrt{\sqrt{\sqrt{f}}}$ dengan pembulatan ke float terdekat ke bawah yang bisa direpresentasikan. Parameter untuk f adalah sebuah float dalam format 32 bits exponent dan 0 bit mantissa.
   b. Untuk mengerjakan soal ini, kita perlu memahami bagaimana mesin menyimpan dan merepresentasikan float. Representasi biner dari suatu bilangan float (*single-precision*) dibagi menjadi 3, yakni:
      i. Sign: 1 bit
      ii. Exponent: 8 bits
      iii. Mantissa: 23 bits
   c. Secara matematis, nilai dari suatu bilangan float yang *normalized* adalah sebagai berikut:
      $$f = (-1)^{sign} + 1.mantissa + 2^{exponent-127}$$
      Karena parameter float pada soal hanya terdiri dari 32 bits exponent, maka sign dan mantissa dapat diabaikan pada soal ini sehingga yang tersisa adalah:
      $$f = 2^{exponent-127}$$
   d. Soal meminta kita untuk mencari nilai dari $\sqrt{\sqrt{\sqrt{f}}}$, dapat ditulis ulang menjadi $f^{\frac{1}{8}}$ sehingga:
      $$f^{\frac{1}{8}} = (2^{exponent-127})^{\frac{1}{8}} = 2^{\frac{(exponent-127)}{8}}$$
      Kita dapat mengaproksimasi nilai dari $\frac{exponent-127}{8}$ dengan menghitung nilai dari $\frac{exponent}{8}$. Walau nilai keduanya tidak sama, pendekatan ini cukup efektif dalam mengaproksimasikan nilai dari ekspresi tersebut.
   e. Nilai dari exponent dibagi dengan 8 dapat dihitung dengan menggunakan operasi right shift sebanyak 3 bits terhadap exponent atau dalam kasus ini f. Ingat bahwa:
      $$a \gg n = \frac{a}{2^n}$$
      Sifat dari operasi right shift selalu membulatkan ke negative infinity atau dalam kata lain membulatkan ke bawah. Sehingga solusinya cukup dalam satu operasi yakni (f >> 3).

3. Referensi
   Wikipedia
   https://en.wikipedia.org/wiki/Single-precision_floating-point_format

Soal 5 – snow_mix

1. Solusi

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

2. Penjelasan
   a. Soal ini meminta praktikan untuk mengembalikan nilai dari $N + 2^{23}$ dengan ketentuan bahwa nilai N:
   $$0 \leq N < 2^{24}$$
   b. Nilai dari 2^{23} (00000000 01000000 00000000 00000000) dapat dihitung dengan menggunakan operasi left shift pada 1 sebanyak 23 bits. Ingat bahwa:
   $$a \ll n = a \times 2^n$$
   Sehingga didapatkan $2^{23} = 1 << 23$ dan disimpan di dalam variabel x.
   c. Langkah selanjutnya adalah menjumlahkan N dengan x. Perhatikan bahwa jika dijumlahkan bit-nya satu per satu, nilai bit yang mungkin terpengaruh hanyalah di bit ke-23. Representasi biner dari 2^{23} adalah 00000000 01000000 00000000 00000000 ($1 \cdot 2^{23} + 0 + 0 + 0 + \ldots$). Karena semua bit kecuali bit ke-23 bernilai 0, nilai N pada semua bit kecuali bit ke-23 sudah terjamin akan sama.
   d. Jika bit ke-23 dari N bernilai 1, maka nilainya akan berubah. Nilai 1 sebelumnya dibawa ke bit selanjutnya yakni bit ke-24. Nilai dari bit ke-24 dan seterusnya sudah terjamin 0 karena ketentuan yang diberikan soal yakni $0 \leq N < 2^{24}$. Perhatikan bahwa untuk $n > 1$ dan $k > 1$:
   $$n^k > n^{k-1} + n^{k-2} + n^{k-3} + \cdots . + n$$
   Karena sifat ini, sudah terjamin bahwa bit ke-24 pasti bernilai 0.
   e. Untuk mengimplementasikannya dalam fungsi, pertama kita akan menghitung nilai dari penjumlahan antara N dan x tanpa nilai bawaan atau *carry*-nya. Hal ini dilakukan dengan menggunakan operasi bitwise XOR pada N dan x. Jika nilai N pada bit ke-23 adalah 0, maka operasi XOR akan mengubahnya menjadi 1 sementara jika nilainya 1, maka operasi XOR akan mengubahnya menjadi nol. Bit lainnya tidak akan berubah.
   f. Selanjutnya, kita menghitung *carry*-nya dengan menggunakan operator AND antara N dan x. Jika nilainya sama-sama 1 pada bit ke-23, maka operator AND akan mengembalikan 1 pada bit ke-23. Hasil dari operasi AND ini selanjutnya akan digeser ke kiri sebanyak 1 bit.

g. Langkah terakhir adalah melakukan operasi XOR pada nilai penjumlahan tanpa bawaan dengan nilai bawaan tersebut. Ini hanya akan mempengaruhi bit ke-24 yang sudah terjamin 0 sehingga terjamin bahwa tidak akan menciptakan nilai bawaan atau *carry* yang baru. Jika ada nilai yang akan dibawa ke bit selanjutnya, maka bit ke-24 akan berubah menjadi 1.

3. Referensi
   Tidak ada

Soal 6 – sky_hundred

1. Solusi

```
int sky_hundred(int n){
    int mod = n & 3;
    return ~(n >> 31)
           & ((n & (~(!(mod ^ 0)) + 1))
           | (1 & !(mod ^ 1))
           | ((n + 1) & (~(!(mod ^ 2)) + 1)));
}
```

2. Penjelasan
   a. Soal ini meminta praktikan mengembalikan hasil operasi XOR dari 1 ke n. Jika nilai dari integer n negatif, maka dikembalikan 0.
   b. Nilai akhir yang dihasilkan dari fungsi dapat dihitung dengan metode berikut:
      i. Jika n mod 4 bernilai 0, maka hasil XOR dari 1 ke n adalah n.
      ii. Jika n mod 4 bernilai 1, maka hasil XOR dari 1 ke n adalah 1.
      iii. Jika n mod 4 bernilai 2, maka hasil XOR dari 1 ke n adalah n + 1.
      iv. Jika n mod 4 bernilai 3, maka hasil XOR dari 1 ke n bernilai 0.
   c. Untuk mengetahui alasan mengapa metode ini bisa digunakan, kita harus mengetahui sifat-sifat dari operasi XOR berikut:
      i. Operasi XOR bersifat komutatif dan asosiatif sehingga urutan operasi tidak berpengaruh pada hasil akhir.
      ii. Untuk setiap bilangan n, $n \wedge n = 0$.
      iii. Untuk setiap bilangan n, $n \wedge 0 = n$.
   d. Perhatikan bahwa jika untuk setiap n bilangan genap (bit pertamanya bernilai 0), nilai dari $n \wedge (n+1) = 1$. Ini karena perubahan hanya terjadi pada bit pertamanya. Untuk operasi XOR dari 1 ke n, kita dapat menuliskannya sebagai berikut:
      $$1 \wedge 2 \wedge 3 \wedge 4 \wedge 5 \wedge \ldots \wedge n$$
      Karena $0 \wedge 1 = 1$, maka bisa juga ditulis sebagai:
      $$0 \wedge 1 \wedge 2 \wedge 3 \wedge 4 \wedge 5 \wedge \ldots \wedge n$$
   e. Berdasarkan observasi sebelumnya, kita dapat mengelompokkannya menjadi $\left\lfloor \frac{n+1}{2} \right\rfloor$ pasangan seperti:
      $$0 \wedge 1 = 1$$
      $$2 \wedge 3 = 1$$
      $$4 \wedge 5 = 1$$
      dan seterusnya.

f. Perhatikan bahwa jika n bernilai genap, maka akan terdapat suatu bilangan yang belum memiliki pasangan yakni n. Namun jika n bernilai ganjil, maka akan semua bilangan akan sudah terpasangkan. Agar lebih jelas perhatikan contoh berikut dengan n adalah bilangan genap.
   Jika n – 1 (ganjil):
   $$(0 \wedge 1) \wedge (2 \wedge 3) \wedge \dots \wedge ((n-2) \wedge (n-1))$$
   Jika n (genap):
   $$(0 \wedge 1) \wedge (2 \wedge 3) \wedge \dots \wedge ((n-2) \wedge (n-1)) \wedge n$$
   Dengan memanfaatkan sifat XOR $n \wedge n = 0$ dan $n \wedge 0 = n$, kita dapat mengobservasi bahwa jika n – 1 (ganjil), maka hasil dari operasi tersebut hanya bisa bernilai 0 atau 1. Sementara jika n (genap), maka hasil dari operasi tersebut adalah hasil XOR dari n dengan hasil XOR dari 1 ke n – 1 (ganjil). Kita mendapat 4 kemungkinan hasil, yakni 0, 1, n ^ 0, atau n ^ 1.
g. Saat jumlah pasangan ganjil, maka hasil XOR dari 1 ke n – 1 (ganjil) akan menghasilkan 1. Sementara saat jumlah pasangan genap, maka maka hasil XOR dari 1 ke n – 1 (ganjil) akan menghasilkan 0. Perhatikan bahwa:
   i. $\left\lfloor \frac{n+1}{2} \right\rfloor$ bernilai ganjil jika n mod 4 bernilai 0 atau n mod 4 bernilai 3.
   ii. $\left\lfloor \frac{n+1}{2} \right\rfloor$ bernilai genap jika n mod 4 bernilai 1 atau n mod 4 bernilai 2.
h. Kita sandingkan dengan kesimpulan pada langkah sebelumnya dan mendapatkan bahwa:
   i. Jika n mod 4 bernilai 0, maka hasil XOR dari 1 ke n adalah n ^ 0 yang sama dengan n.
   ii. Jika n mod 4 bernilai 1, maka hasil XOR dari 1 ke n adalah 1.
   iii. Jika n mod 4 bernilai 2, maka hasil XOR dari 1 ke n adalah n ^ 1 yang sama dengan n + 1 karena n genap.
   iv. Jika n mod 4 bernilai 3, maka hasil XOR dari 1 ke n bernilai 0.
i. Karena sudah terbukti bahwa metode ini dapat diaplikasikan, kita perlu mengimplementasikannya dalam kode program. Pertama kita right shift n sebanyak 31 bits. Karena mesin diasumsikan melakukan right shift secara aritmetik, maka mesin akan mempertahankan tanda bilangan sehingga jika bilangan bertanda negatif, maka menjadi 0xFFFFFFFF (11111111 11111111 11111111 11111111) dan jike bernilai positif menjadi 0x0 (00000000 00000000 00000000 00000000).

j. Karena tidak diperbolehkan menggunakan conditionals, maka kita bisa menggunakan ini sebagai mask. Namun, kita ingin mempertahankan nilai dari operasi-operasi lainnya jika n nonnegatif sehingga (n >> 31) dioperasikan dengan bitwise NOT menjadi ~(n >> 31) untuk membalikkan nilainya. Jika n nonnegatif, maka mask akan bernilai 0xFFFFFFFF.
k. Untuk mengecek berapa nilai n mod 4, kita menggunakan operasi AND (n & 3). Operasi n mod 4 memberikan sisa dari n div 4. Saat melakukan operasi AND, kita mematikan semua bit selain dengan 2 bits pertama. Nilai yang didapat akan ekivalen dengan n mod 4.
l. Untuk mengecek apakah nilai dari n mod 4 bernilai 0, 1, 2, atau 3, kita menggunakan sifat dari operasi XOR yaitu $n \wedge n = 0$. Untuk mengeceknya, kita melakukan XOR antara nilai mod dengan nilai yang akan diperiksa. Setelah itu, kita operasikan dengan logical NOT sehingga dia mengembalikan 1 jika nilainya sama dan mengembalikan 0 jika nilainya tidak sama. Selanjutnya, kita membuat mask bagi nilai n dari hasil yang kita dapat tersebut. Ini dapat dilakukan dengan melakukan operasi bitwise NOT lalu ditambahkan dengan 1. Jika nilai mod sama, maka mask akan menjadi 11111111 11111111 11111111 11111111 dan jika tidak sama maka menjadi 00000000 00000000 00000000 00000000.
m. Kita tinjau keempat kasus tersebut:
    i. Kasus n mod 4 = 0
       Untuk memeriksa apakah nilai n mod 4 = 0, kita lakukan operasi !(mod ^ 0) lalu nilai tersebut diubah menjadi mask. Kita kemudian menggunakan operasi AND untuk mengembalikan n jika n mod 4 = 0 dan mengembalikan 0 jika n mod 4 bukan 0.
    ii. Kasus n mod 4 = 1
       Berbeda dengan kasus n mod 4 = 1, kasus ini mengembalikan suatu nilai yang tidak bergantung pada n yakni 1 saja. Artinya kita tidak perlu menggunakan mask dan cukup memeriksa nilainya dengan 1 & !(mod ^ 1).
    iii. Kasus n mod 4 = 2
       Kasus ini mirip dengan kasus saat n mod 4 = 0. Yang berbeda hanyalah nilai yang dioperasikan dengan XOR dan nilai pengeluarannya. Kita hanya memeriksa apakah nilai dari n mod 4 = 2 dengan !(mod ^ 2), lalu nilai tersebut diubah menjadi mask untuk n + 1.

iv. Kasus n mod 4 = 3
Kasus ini tidak tertulis secara eksplisit di dalam kode. Hal ini disebabkan nilai yang dikembalikan adalah 0. Karena pengecekan sebelumnya juga akan mengembalikan 0 jika tidak terpenuhi, maka penambahan pengecekan (0 & !(mod ^ 3)) dianggap redundan.

n. Langkah terakhir adalah menggabungkan keempat kasus di atas dengan operasi bitwise OR. Operasi tersebut akan menghasilkan keluaran untuk nilai n nonnegatif. Setelah itu, perlu dipastikan nilainya positif melalui operasi AND dengan mask awal (~(n >> 31)) untuk mendapatkan jawabannya.

3. Referensi
Computer Science Stack Exchange
https://cs.stackexchange.com/questions/157353/calculating-xor-of-all-numbers-from-1-to-n-why-does-this-method-work

Soal 7 – ganganji

1. Solusi

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

2. Penjelasan
    a. Soal ini meminta praktikan mengalikan nilai integer x dengan 1.125 dan dibulatkan ke bawah. Nilai x terjamin nonnegatif dan jika terjadi overflow, maka dikembalikan 0x7FFFFFFF.
    b. Untuk menghitung nilai dari 1.125 * x, kita pertama dapat menyederhanakan ekspresi 1.125. Jika diubah ke bentuk pecahan, maka $1.125 = \frac{9}{8} = 1\frac{1}{8} = 1 + \frac{1}{8}$. Karena tidak diperbolehkan menggunakan operasi perkalian (*), kita dapat menggunakan operasi right shift.

$$x * 1.125 = x(1 + \frac{1}{8}) = x + \frac{x}{8}$$

Ingat bahwa $a \gg n = \frac{a}{2^n}$ dan $\log_2 8 = 3$ sehingga:

$$x + \frac{x}{8} = x + \frac{x}{2^3} = x + (x >> 3)$$

Nilainya kita simpan di variabel integer xtimed.
    c. Karena tidak diperbolehkan menggunakan konstanta besar, kita memerlukan cara lain untuk mencari 0x7FFFFFF (01111111 11111111 11111111 11111111). Ini dapat dilakukan dengan menggeser 1 ke kiri sebanyak 31 bits (1 << 31) membentuk 0x80000000 (10000000 00000000 00000000 00000000) lalu menggunakan operasi bitwise NOT untuk membalikkan nilai 0 dan 1 sehingga nilainya menjadi 0x7FFFFFFF (10000000 00000000 00000000 00000000).
    d. Untuk mengecek apakah overflow terjadi, cukup dilakukan dengan mengecek sign (bit ke-31) dari nilai xtimed. Karena nilai x sudah terjamin nonnegatif, sign xtimed yang bernilai negatif (bit ke-31 bernilai 1) akan menandakan bahwa terjadi overflow. Pengecekan dilakukan dengan melakukan operasi right shift sebanyak 31 bits. Karena mesin melakukan right shift secara aritmetik, maka jika nilai sign adalah 1 maka nilai dari variabel overflow ini adalah 0xFFFFFFFF (11111111 11111111 11111111 11111111) dan jika nilai sign adalah nol, maka nilainya adalah 0x0 (00000000 00000000 00000000 00000000). Nilai dari integer overflow ini akan digunakan sebagai mask dalam langkah selanjutnya.

e. Kita mengembalikan nilai (overflow & max_int) | (~overflow & xtimed). Nilai dari overflow dijadikan sebagai mask untuk max_int dan xtimed. Jika overflow bernilai 0xFFFFFFFF, maka ini menandakan terjadi overflow dan dengan operasi AND akan mengembalikan nilai dari max_int sementara nilai overflow dioperasikan dengan bitwise NOT menjadi 0x0 lalu nilai xtimed dioperasikan AND dengan nilai 0x0 tersebut. Ini secara efektif mematikan nilai dari xtimed sehingga setelah dilakukan operasi OR antara (overflow & max_int) dengan (~overflow & xtimed) hanya akan dikembalikan nilai max_int jika overflow terjadi. Sebaliknya jika overflow tidak terjadi, maka nilai variabel overflow akan menjadi 0x0 dan fungsi akan mengembalikan nilai xtimed.

3. Referensi
   Tidak ada

Soal 8 – kitsch

1. Solusi

```
int kitsch(int x) {
    int divBy64 = x >> 6;
    int remainder= x & 63;
    int multBy17 = (divBy64 << 4) + divBy64;
    int remainderMult = (remainder << 4) + remainder;
    return multBy17 + ((remainderMult + ((x >> 31) & 63)) >> 6);
}
```

2. Penjelasan
   a. Soal ini meminta praktikan mengembalikan nilai $x * \frac{17}{64}$. Hasilnya dibulatkan menuju 0.
   b. Walaupun nilai dari x * $\frac{17}{64}$ lebih kecil dari nilai x, kita tidak bisa langsung saja mengalikan x dengan 17 lalu membagi hasilnya dengan 64. Hal itu disebabkan tidak ada yang mencegah hasil dari x * 17 untuk mengalami overflow. Karena itu, kita harus membaginya dengan 64 terlebih dahulu agar mencegah overflow. Ingat bahwa:
   $$a \gg n = \frac{a}{2^n}$$
   Karena $\log_2 64 = 6$, maka $\frac{x}{2^6} = \frac{x}{64} = x >> 6$. Untuk memastikan perhitungannya lebih akurat, kita simpan nilai yang tidak habis terbagi di variabel remainder. Ini biasanya dilakukan dengan melakukan n mod 64, namun karena operasi modulo tidak diperbolehkan, maka kita harus menggunakan operasi AND sehingga remainder = n & 64 – 1 = n & 63.
   c. Sekarang, kita dapat mengalikan kedua nilai di bagian sebelumnya dengan 17. Perhatikan bahwa:
   $$divBy64 * 17 = divBy64(16 + 1) = 16divBy64 + divBy64$$
   Ingat bahwa:
   $$a \ll n = a \times 2^n$$
   Karena nilai $\log_2 16 = 4$, nilai dari divBy64 * 17 adalah:
   $$divBy64 * 17 = 16 * divBy64 + divBy64 = (divBy64 << 4) + divBy64$$
   Dengan melakukan operasi yang sama pada remainder, kita mendapat remainderMult = (remainder << 4) + remainder.
   d. Hal penting yang perlu diketahui adalah cara pembulatan melalui operasi right shift dan pembulatan melalui division biasa dalam bahasa C. Jika melakukan division biasa, maka nilainya akan dibulatkan ke 0 sementara jika melalui shifting akan dibulatkan ke $-\infty$. Ini tidak menjadi masalah untuk bilangan positif tetapi akan menjadi isu untuk bilangan negatif. Cara mengatasinya adalah menambahkan suatu bias jika nilainya negatif.

e. Bias ini dihitung dengan pertama memeriksa nilai dari sign. Ini dilakukan dengan right shift 31 bits (x >> 31). Karena aritmethic shift, dia akan menghasilkan 0xFFFFFFFF jika nilai sign 1 dan menghasilkan 0x0 jika nilai sign 0. Agar tidak mempengaruhi nilai lain, kita batasi nilainya ke 6 bits pertama melalui operasi AND dengan 63. Kita membatasinya dengan 6 bits pertama karena nilai dari remainder belum dibagi dengan 64 sehingga biasnya harus bernilai 64 – 1 agar dapat memberikan pengaruh. Setelah itu, hasil penjumlahan remainderMult dan bias dibagi dengan 64 melalui operasi right shift sebanyak 6 bits. Ini secara efektif mengembalikan nilai dari remainder * 17 / 64 jika nilai positif dan menambahkan bias sebesar 1 jika nilai negatif.
f. Hasil akhirnya didapat dengan menjumlahkan nilai multBy17 yang merupakan hasil (x div 64) * 17 dengan hasil dari (remainder * 17) div 64 setelah mempertimbangkan bias.

3. Referensi

Stack Overflow
https://stackoverflow.com/questions/42127446/bit-manipulation-understanding-rounding-toward-zero-bias-when-multiplying-a-neg

Soal 9 – how_to_sekai_seifuku

1. Solusi

```
unsigned how_to_sekai_seifuku(unsigned f) {
    unsigned sign = (f >> 15) & 0x1;
    unsigned exponent = (f >> 10) & 0x1F;
    unsigned mantissa = f & 0x3FF;
    unsigned result = 0;

    if (exponent == 0x1F) {
        if (mantissa == 0) {
            if (sign == 0x1) {
                result = 0xFF800000;
            } else {
                result = 0x7F800000;
            }
        } else {
            result = 0x7F800001;
        }
    } else if (exponent == 0) {
        if (mantissa == 0) {
            result = (sign << 31);
        } else {
            exponent = 0x71;
            while ((mantissa & 0x400) == 0) {
                mantissa = mantissa << 1;
                exponent = exponent - 1;
            }
            mantissa = mantissa & 0x3FF;
            result = (sign << 31) | (exponent << 23) | (mantissa << 13);
        }
    } else {
        exponent = exponent + 0x70;
        result = (sign << 31) | (exponent << 23) | (mantissa << 13);
    }
    return result;
}
```

2. Penjelasan

a. Soal ini meminta praktikan mengubah suatu bilangan half-precision floating point menjadi single-precision floating point dengan penanganan kasus +/- inf, +/- 0, dan NaN (harus mengembalikan 0x7F800001.

b. Langkah pertama adalah mengekstrak nilai sign, exponent, dan mantissa dari f. Dalam half-precision floating point format, sign terdiri dari 1 bit, exponent terdiri dari 5 bits, dan mantissa terdiri dari 10 bits. Nilainya diekstrak dengan cara right shift lalu menggunakan operasi AND dengan suatu mask.

   i. sign = (f >> 15) & 0x1 (00000000 00000000 00000000 00000001)

   ii. exponent = (f >> 10) & 0x1F (00000000 00000000 00000000 00011111)

   iii. mantissa = f & 0x3FF (00000000 00000000 00000011 11111111)

c. Kemudian kita tangani kasus khusus dimana exponent bernilai 0x1F atau semua bit-nya bernilai 1. Saat semua

bit dalam exponent bernilai 1, maka ada dua kemungkinan yakni nilai adalah +/- inf (tergantung sign) atau nilai adalah NaN. Jika saat exponent semuanya bernilai 1 sementara mantissa bernilai 0, maka nilai tersebut adalah nilai infinity. Karena diperbolehkan konstanta besar, kita dapat langsung menyimpan nilai dari +inf (0x7F800000) atau -inf (0xFF800000). Jika nilai mantissa bukan 0, maka nilai tersebut termasuk NaN dan nilai yang disimpan adalah 0x7F800001.

d. Selanjutnya, kita periksa kasus dimana exponent bernilai 0. Pertama periksa terlebih dahulu apakah mantissa-nya juga bernilai 0. Jika iya, maka nilainya menjadi +/- 0 tergantung pada sign sehingga sign akan digeser kembali ke posisi awalnya melalui (sign << 31). Namun jika tidak, maka bilangan ini disebut denormalized number. Nilai dari exponent harus dikonversikan terlebih dahulu.

$$e = exponent\ yang\ disimpan\ (half)$$
$$e' = exponent\ yang\ disimpan\ (single)$$
$$E = exponent\ sebenarnya (half)$$
$$E' = exponent\ sebenarnya\ (single)$$

   i. $E = e - 15$
   ii. $E' = e' - 127$

   Kita dapat mengonversinya menjadi: $e' = E - 15 + 127 = E + 112$. Karena angka denormalized, maka exponent yang disimpannya disimpan menjadi 113 (atau 1 + 112).

e. Nilai mantissa kemudian dinormalisasi dengan cara menggeser nilai dari mantissa hingga bit ke-10nya bernilai 1. Ini dilakukan melalui operasi AND dengan mask 0x400 (00000000 00000000 00000100 00000000). Selama masih 0, maka dia akan terus menggeser nilai mantissa ke kiri (mantissa << 1) sementara mendecrement nilai dari exponent. Saat nilai 1 ditemukan di bit ke-10, maka loop akan berhenti. Perhatikan bahwa nilai 1 yang ditemukan tidak disimpan secara eksplisit pada mantissa namun sebagai implicit leading 1.

f. Nilai dari sign, exponent, dan mantissa pun diletakkan pada tempatnya di representasi single precision floating point dimana sign berada pada bit ke-31, exponent berada pada bit k-23 hingga bit ke-30 sementara mantissa berada pada bit ke-0 hingga bit ke-22.

   i. (sign << 31)
   ii. (exponent << 23)
   iii. (mantissa << 13) (Ini karena mantissa menempel ke kiri)

   Semuanya digabung dengan operasi bitwise OR.

    g. Namun, jika bilangan dalam half-precision sudah normalized, kita hanya perlu mengubah exponent-nya saja. Seperti yang ada persamaan tadi, exponent yang disimpan di half precision ditambahkan dengan 112 (exponent + 0x70). Kemudian nilai sign, exponent, dan mantissa diletakkan pada tempatnya sama seperti cara sebelumnya.
    h. Hasil akhir akan mengembalikan representasi biner dari bilangan half precision floating point f dalam representasi single precision floating point.
3. Referensi

Wikipedia
https://en.wikipedia.org/wiki/Half-precision_floating-point_format
https://en.wikipedia.org/wiki/Single-precision_floating-point_format
Stack Overflow
https://stackoverflow.com/questions/71120169/bit-shifting-a-half-float-into-a-float

Soal 10 – mesmerizer

1. Solusi

```
int mesmerizer(unsigned uf) {
    unsigned sign = (uf >> 31) & 0x1;
    unsigned exponent = (uf >> 23) & 0xFF;
    unsigned mantissa = uf & 0x7FFFFF;
    int result = 0;

    if (exponent == 0xFF) {
        result = 0x80000000u;
    } else {
        if (exponent >= 127) {
            exponent = exponent - 127;
            if (exponent <= 31) {
                result = (mantissa | 0x800000) >> (23 - exponent);
                if (sign) {
                    result = -result;
                }
            } else {
                result = 0x80000000u;
            }
        } else {
            result = 0;
        }
    }
    return result;
}
```

2. Penjelasan
    a. Soal ini meminta praktikan untuk mengembalikan bit-level equivalent dari ekspresi (int) f. Jika float melebihi batasan, maka kembalikan 0x80000000u.
    b. Langkah pertama adalah mengekstrak nilai sign, exponent, dan mantissa dari f. Dalam single-precision floating point format, sign terdiri dari 1 bit, exponent terdiri dari 8 bits, dan mantissa terdiri dari 23 bits. Nilainya diekstrak dengan cara right shift lalu menggunakan operasi AND dengan suatu mask.
        i. sign = (f >> 31) & 0x1 (00000000 00000000 00000000 00000001)
        ii. exponent = (f >> 23) & 0x1F (00000000 00000000 00000000 11111111)
        iii. mantissa = f & 0x3FF (00000000 01111111 11111111 11111111)
    c. Kita akan memeriksa nilai exponent terlebih dahulu. Jika nilai exponent-nya 0xFF (00000000 00000000 00000000 11111111), maka itu menandakan nilai tersebut NaN atau infinity. Karena konstanta besar diperbolehkan, langsung saja mengembalikan nilai 0x8000000u.
    d. Kita kemudian akan memeriksa apakah float tersebut termasuk normalized number. Pengecekan ini dilakukan dengan memeriksa apakah nilai dari exponent ≥ 127. Jika iya, maka nilai exponent sebenarnya diekstrak. Nilai

exponent sebenarnya didapat dari mengurangi exponent yang disimpan dengan bias yang pada single precision floating point berupa 127. Jika nilai exponent berada pada range yang valid, maka kita menaruh implicit leading 1 nya dengan menggunakan (mantissa | 0x800000) dan menyesuaikan mantissa dengan exponent. Ini dilakukan dengan melakukan right shifting sebanyak (23 - nilai exponent) bits. Ini harus dilakukan karena exponent mengandung nilai yang perlu dipertimbangkan dalam casting. Jika ada tanda negatif, maka nilai akan menjadi versi negatifnya. Operator - diperbolehkan sehingga menggunakan itu saja.

e. Jika mantissa tidak berada pada range yang valid, maka itu melebihi batasan sehingga kembalikan 0x8000000u.
f. Langkah terakhir memastikan bahwa jika exponent yang disimpan lebih kecil dari 127 akan mengembalikan 0 karena angkanya mendekati nol. Hasilnya dikembalikan di akhir fungsi.

3. Referensi
   Tidak ada