$\text{Computersystemen}_2016_1$

Micha de Groot - 10434410 & Jorgos Tsovilis - 10246878

September 2016

Puzzle 1 - bitOr

Maak een operator die een OR operatie uitvoerd op twee variabelen. Er mag alleen gebruik gemaakt worden van de volgende operators: ~, &.
Hier is door wat passen en meten een vrij simpele oplossing gevonden. Door eerst de negatie te nemen van de beide variabelen, vervolgens een AND operatie hier op uit te voeren en van deze uitkomst weer de negatie te nemen is het zeker dat als tenminste een van de twee bits een 1 bevat dat dan de uitkomst ook een 1 is.
Voorbeeld:
X = 100
Y = 110
~X = 011
~Y = 001
~X & ~Y = 001
~(~X & ~Y) = 110

```
int bitOr(int x, int y) {
    int result = ~(~x & ~y);
    return result
```

Puzzle 2 - minusOne

Maak een functie die altijd de waarde -1 geeft. Er mag alleen gebruik gemaakt worden van de de volgende operators: !, ~, &, $\wedge$, |, +, <<, >>.
Doordat er geen gebruik gemaakt mag worden van de - operator moet er handmatig een integer gemaakt worden van alleen enen. Dit wordt gedaan door eerst met leftshift een integer te maken met als meest significante bit een één. Daarna kan met rightshift de één weer worden teruggeschoven. Doordat het een arithmetic shift is worden alle bits een één.

```
int minusOne(void) {
    int integer = 1 << 31;
    integer = integer >> 31;
```

```
    return integer;
```

Puzzle 3 - allOddBits

Maak een functie die 1 geeft als alle even (of oneven?) bits 1 zijn, anders 0. Er mag alleen gebruik gemaakt worden van de de volgende operators: !, ~, &, ∧, |, +, <<, >>.
Voor deze opgave is een mask nodig van een integer met op alle even plaats enen, 0xAAAAAAAA, om mee te vergelijken. Aangezien er maximaal een integer van grootte 0xFF aagemaakt kan worden, moet de mask gemmakt worden door een aantal shifts en OR operaties. De vergelijking tussen de input en de mask wordt gedaan door eerst een AND operatie te doen. Hierdoor blijven alle enen die op even plekken staan over. Daarna kan er door XOR operatie te doen een integer van nullen gemaakt worden, tenzij er in de originele input op een even bit een nul stond. Aan het einde wordt een logische negatie gedaan om een één als output te geven als er na de XOR operatie alleen nullen waren, anders is de output een nul.

```
int allOddBits(int x) {
    int mask = 0xAA;
    int temp = 0xAA;
    temp = temp << 8;
    mask = temp | mask;
    temp = temp << 8;
    mask = temp | mask;
    temp = temp << 8;
    mask = temp | mask;
    int number = x & mask;
    number = number ^ mask;
    return ! number;
}
```

Puzzle 4 - isAsciiDigit

Maak een functie die 1 terug geeft als de integer x tussen of gelijk is aan 0x30 en 0x39, en geef anders 0 terug. Bijvoorbeeld: isAsciiDigit(0x35) = 1, isAsciiDigit(0x3a) = 0.
Als eerste wordt er een int(sign) gemaakt die alleen een 1 heeft op het sign bit. Door een OR operatie te doen met deze sign en de hoogst mogelijke waarde(0x39) en hier de negatie van te nemen ontstaat er het volgende bit: 01111...11000110 dit is de upperbound. Als de waarde van x hoger is dan 0x39 en word opgeteld bij de upperbound zal blijken dat de linker enen van upperbound allemaal nullen worden en het laatste bit een 1. In gevallen waar de waarde van x lager of gelijk is aan 0x39 zal het meest linker bit 0 blijven.

Voor de lowerbound geldt, door de negatie van 0x30 te nemen(11111...11001111) dat hier de linker bits nullen worden als er iets bij op word geteld dat hoger of gelijk is aan 0x30. In andere gevallen blijven deze linker bits enen.
Dus als deze upperbound vervolgens 31 bits naar rechts worden geshift blijven er alleen enen over als de waarde van x boven 0x39 ligt. Voor de lowerbound geldt dat er alleen enen over blijven als x onder 0x30 ligt. Door uiteindelijk de OR operatie te doen met de upper en lowerbound zal er in gevallen waar één van deze twee een één was een 1 uit komen. Door hier de logical NOT operatie op toe te passen zal er 0 uitkomen in gevallen buiten 0x30 en 0x39, en in gevallen binnen deze getallen een 1.

```
int isAsciiDigit(int x) {
    int sign = 1 << 31;
    int upperbound = ~(sign | 0x39);
    int lowerbound = ~0x30;

    upperbound = sign & (upperbound+x) >> 31;
    lowerbound = sign & (lowerbound+1+x) >> 31;
  return !(upperbound | lowerbound);
}
```

Puzzle 5 - leftBitcount

Maak een functie telt hoeveel enen er achter elkaar staan vanaf de meest linker bit. Bijvoorbeeld: leftBitCount(-1) = 32, of leftBitCount(0XF0000000) = 4.
De functie werkt door eerst naar de eerste 16 bits te kijken, als deze allemaal 1 zijn dan word er een 1 gezet op de vijfde bit(16) in result(res). Daarna word er door gegaan met de overige bits door eerst te kijken naar de volgende 8 bits, daarna 4 bits, dan 2 en uiteindelijk het laatste bit. In het geval dat de bits die worden gecontroleerd(16, 8, 4, 2 en 1) niet allemaal 1 zijn word er een 0 toegevoegd op de bijbehorende bit plaats in result. Als er bij 16, 8, 4 of 2 niet enkel enen worden gevonden dan word er de andere kant op gekeken, dus "zijn van de eerste 16 bits de eerste 8 bits wel enkel enen?".

```
int leftBitCount(int x) {
    int res;
    int tmp = x;
    int shft;
    // Shift by 16 bits to check if we are left with one or zero.
    // If the first 16 bits are ones we will be left wih a one.
    // If not then we're left with a zero.
    res = !(~(tmp>>16)) << 4;
    tmp = tmp << res;
    // Shift by 24 bits to check the remaining 8 bits.
    shft = !(~(tmp >> 24)) << 3;
```

```
    tmp = tmp << shft;
    res = res | shft;
    // Shift by 28 bits to check the remaining 4.
    shft = !(~(tmp>>28)) << 2;
    tmp = tmp << shft;
    res = res | shft;
    // check last two bits.
    shft = !(~(tmp >> 30)) << 1;
    tmp = tmp << shft;
    res = res | shft;
    // check last bit.
    res= res ^ (1&((tmp>>31)));
    // Add one to the result to have 32.
    return res + !(~x);
}
```

Puzzle 6 - float_f2i

Maak een functie die de input variabele (een 32-bit int) interpreteert als 32-bit float en de waarde daarvan als int teruggeeft. Belangrijk hierbij is om te weten hoe een float is opgebouwd; het eerste bit is het teken, de volgende acht zij het gecodeerde exponent en de laatste 23 bits de fractie. Deze elementen moeten met behulp van masks uit de input gehaald worden. Het exponent wordt gedecodeerd door er de bias (in dit geval 127) af te halen.
Vervolgens zijn er enkele edge-cases waarvoor een speciale output gegeven moet worden: Als het gecodeerde exponent 0xFF is dan moet de float gezien worden als oneindig en wordt er 0x80000000u teruggegeven. Bij een gecodeerde exponent van 0 is de totale waarde altijd minder dan één en is het eindresultaat nul.Idem dito voor het gedecodeerde exponent. Ook als het exponent groter is dan 30 valt het buiten de mogelijke range.
Dan komt nu de berekening voor de reguliere getallen. Hier worden twee gevallen onderscheden. Aangezien de fraction eigenlijk hetgeen is dat na de komma staat moet er gecorrigeerd worden voor het feit dat de fractie eigenlijk al 23 plaatsen naar links is geshift. De hoeveelheid die de fractie door het exponent geshift moet worden wordt dus minder. Als het exponent groter of gelijk is aan 23 moet er 23 keer minder naar links geshift worden. Is het exponent kleiner dan moet de fractie weer wat naar rechts worden geshift. Daarna moet het getal eventueel negatief gemaakt worden als het sign bit één is. Dit wordt gedaan door de inverse te nemen en er één bij op te tellen.

```
int float_f2i(unsigned uf) {
// split the sign, exponent and frac from the input
    int mask = 0x80000000;
    int sign = uf & mask;
    int bias = 127;
```

```
    int exp, e, frac;
    mask = 0x7F800000;
    exp = (mask & uf) >> 23;
    e = exp - bias;
    mask = 0x007FFFFF;
    frac = uf & mask;

// Several edge cases
    if(exp == 0x7F800000){
        return 0x80000000u;
    }
    if(!exp){
    return 0;
    }
    if(e<0){
    return 0;
    }
    if(e >30){
    return 0x80000000u;
    }

// Compensate for the fact that the frac is already shifted 23 times
    frac = frac | 0x800000;
    if(e >= 23){
        frac = frac << (e-23);
    }else{
    frac = frac >> (23-e);
    }
// Add sign
    if(sign){
        frac = ~frac+1;
    }

    return frac;
}
```