CSED211 Intro. to Computer SW Systems : Lab2 Report

September 16, 2025

Student 20240505 Hyunseong Kong

1. Overview

This lab aims to solve the 6-consecutive bit puzzle. Two of the problems are related to the two's complement of a 32-bit integer, and the rest are related to floating point, manipulating values at the bit level.

2. Implementation

2-1. negate(x)

Request: Given `x`, compute $-x$.

```
int negate(int x) {
    return (~x)+1;
}
```

Explanation: This is straightforward in terms of two's complement.
First, the bitwise NOT operator $\sim$ flips each bit. This directly implies `x + (~x) = 0xFFFFFFFF`. If we add `0x00000001`, then each bit is carried in its original position. Since the carry-out occurs at the 33rd bit, it is discarded. Therefore, the whole addition results in `0x00000000`. Hence, the additive inverse of `x` is `(~x)+1`.

2-2. isLess(x, y)

Request: Given `x` and `y`, return 1 if $x < y$, and return 0 otherwise.

```
int isLess(int x, int y) {
    return 0x1 & ((((x ^ y) & (~y)) | ((~(x ^ y)) & ~(y+(~x)))) >> 31);
}
```

Explanation:

- `sign of x != sign of y`: It suffices to rely on the sign of one of them (say y), because only $x < 0 < y$ or $y < 0 < x$ can hold. If we choose y as the reference, then when $y > 0$ the result is necessarily true, and when $y < 0$ it is necessarily false. Note that the same reasoning applies symmetrically to x.
- `sign of x == sign of y`: We need to compute `y - x - 1`. Since the original condition is `y - x > 0`, in this case we check whether `y - x - 1 >= 0`. In two's-complement arithmetic the additive inverse of `x` is `(~x)+1`, so we can simplify the check by removing the trailing '1' from the computation.

To merge these two situations into a single expression, we use a following selector: `(A & expr1) | ((~A) & expr2)`, where we plug in `A = (x ^ y)`.
Finally, since the decision is carried by the most significant bit, we mask with `0x1 & (□ >> 31)`.

2-3. float_abs(uf)

Request: Given `uf`, compute the absolute value of f.

```
unsigned float_abs(unsigned uf) {
    if ((!((uf & 0x7F800000) ^ 0x807FFFFF)+1)) & (!!(uf & 0x007FFFFF))) return uf;
    return uf & (0x7FFFFFFF);
}
```

Explanation: To satisfy the requirements, we must handle `NaN` separately. We therefore need a way to detect whether `uf` is `NaN`. To do this, check if the exponent field is `0xFF` and the fraction field is nonzero, simultaneously.

1. `Exp Field == 0xFF`: First compute (`uf & 0x7F800000`) to isolate the 8-bit exponent field. Next, XOR it with `0x807FFFFF` ($= 011111111000\ldots00$). Since XOR yields `0xFFFFFFFF` only when the two numbers are exactly the same, adding 1 gives 0 if and only if they match. Interpreting 0 as False in C, we then apply logical NOT `!` to flip it.
2. `Fraction Field != 0`: Compute `uf & 0x007FFFFF` to isolate the 23-bit fraction field. Applying the logical NOT operator twice makes any nonzero values become `0x1` and zero becomes `0x0`. We use `!!` so we can AND this boolean result bitwisely with the previous condition.

If the condition holds(i.e. `NaN`), the code returns `uf` unchanged. Otherwise, it clears the most significant bit with a bitwise AND and returns the result.

2-4. float_twice(uf)

Request: Given `uf`, compute $2 \times f$.

```
unsigned float_twice(unsigned uf) {
    unsigned e = uf & 0x7F800000;
    unsigned s = uf & 0x80000000;
    unsigned m = uf & 0x007FFFFF;
    unsigned ae = e + 0x00800000;
    if (!((e ^ 0x807FFFFF)+1)) return uf;
    if (!e) return s+e+(m << 1);
    if (ae ^ 0x7F800000) return s+ae+m;
    else return s+0x7F800000;
}
```

Explanation: For convenience, we precompute repeated components. Here `s`, `e`, and `m` denote the sign, exponent, and fraction fields, respectively. When returning the value, we combine `s`, `e`, and `m` by addition at their native bit positions, so we avoid extra shifting to these values.

- `Exp == 0xFF`: `NaN` and $\pm\infty$ remain unchanged when doubled. In particular, `NaN` must be returned with unchanged value as it is the problem statement. We therefore return `uf` when the exponent is `0xFF`. (The detection method is the same as in 2-3 so we omit it)
- `Exp == 0x0`: This is a denormalized value. In this case, we simply shift `m` left by one and return it with the original sign. If `m` does not spill into the exponent field, the result is naturally doubled. If it does, the value becomes normalized. In the normalized interpretation, the exponent differs by 1 compared to the denormalized representation, and the hidden leading bit changes from 0 to 1, yielding exactly a factor of 2 overall.
- `(Exp != 0x0) && (Exp != 0xFF)`: This is a normalized value. Doubling it amounts to incrementing the exponent by 1. Since we are not shifting bit positions, we add a value whose 9th bit from the left is 1 to `e`, denoted `ae`. If `ae != 0xFF`, we return `s + ae + m`. If `ae` becomes `0xFF`, the result overflows to infinity. In this case, we return an exponent of `0xFF` and a fraction of `0x0`, preserving the original sign.

2-5. float_i2f(x)

Request: Given `x`, compute `(float)`x.

```
unsigned float_i2f(unsigned x) {
    unsigned s = x & 0x80000000;
    unsigned xn = x;
    unsigned l = 0;
    unsigned r = 0;
    unsigned t = 0;
    unsigned l7 = 0;
    if (s) xn = -x;
    if (x == 0x00000000) return 0x00000000;
    if (x == 0x80000000) return 0xCF000000;
    for (;!((xn<<l) & (0x40000000));l++) {}
    t = xn << l;
    l7 = t & 0x7F;
    r = (l7 > 0x40) || (l7 == 0x40 && ((t >> 7) & 1));
    return s + ((157-l) << 23) + ((t >> 7) & 0x007FFFFF) + r;
}
```

Explanation: For convenience, we first define the values repeatedly appear in the submitted code:

- `l`: The number of left shifts needed to bring the first fraction bit into position.
- `r`: The rounding increment. (Can be either `0x0` or `0x1`)
- `t`: The value after left-shifting by `l`.
- `l7`: The bits of `t` that will be discarded(7 least significant bits) when reserving space for the exponent field.

We first separate the sign and magnitude of `x`, because the sign of a float is represented as $(-1)^s$ rather than by two's complement.
Next, we find the bit that must become the first fraction bit using a `for` loop. Importantly, when storing the fraction, the leftmost `1` is implicit, so the loop's purpose is to locate that `1`. If all bits except the sign are zero, the loop would otherwise be infinite. Since there are exactly two such cases, we handle by returning precomputed values.

- Since `xn` is the absolute value, its most significant bit is guaranteed to be 0. After left-shifting `xn` by `l`, we check against `0x40000000`(the second bit from the left) with an AND to detect the leading `1`. Because a `for` loop checks its condition before executing `l++`, if the second bit is already 1, the loop safely terminates with `l = 0`.

After determining `l`, we compute `t = xn << l`.
`l7` stores the 7 least significant bits of `t`. Since the exponent uses 8 bits, we discard 7 bits on the right. The second bit from the left in `t` becomes the hidden leading 1 in the float and is therefore not kept explicitly, which is why we drop 7(not 8) bits on the right.
Because we discard bits, we must round. Rounding is applied to the fraction field, with the increment stored in `r`. Thus `r` must be either 0 or 1, and using logical OR `||` ensures this.

- `l7 > `$1000000_{(2)}$: The discarded part is large, so we round up. This is ensured by `l7 > 0x40`.
- `l7 == `$1000000_{(2)}$: This is the tie case. We round to even, making the least significant bit of `(t >> 7)` zero if possible. To to this, we check `((t >> 7) & 1)`. If it is 1, the condition is true and we set `r=1`.
- `l7 < `$1000000_{(2)}$: The discarded part is small, so we round down. Neither condition holds and thus `r=0`.

Finally, `l` determines the exponent field. Because `l` shifts within the 30 data bits(excluding the sign bit and the hidden leading 1), and the exponent uses a bias of 127, the exponent becomes `(30 - l) + 127`. We then drop the implicit leading 1 from `t` with `& 0x007FFFFF`, and return the sign, exponent, rounded fraction, and `r` combined.

2-6. float_f2i(uf)

Request: Given `uf`, compute `(int)`f.

```
int float_f2i(unsigned uf) {
    unsigned e = uf & 0x7F800000;
    int ne = (e >> 23) - 127;
    unsigned s = uf & 0x80000000;
    unsigned m = uf & 0x007FFFFF;
    unsigned re = 0;
    if (!((e ^ 0x807FFFFF)+1)) return 0x80000000;
    if (!e) return 0x00000000;
    if (ne < 0) return 0;
    if (ne > 30) return 0x80000000;
    re = ((m << 7) | 0x40000000) >> (30-ne);
    if (s) return -re;
    return re;
}
```

Explanation: Again, we define the repeated components where `e` is the exponent field, `s` the sign field, and `m` the fraction field.
`ne` converts the stored exponent to the unbiased exponent by right-shifting 23 and subtracting the bias 127. We use `int` for `ne` because its range is -127 to 126, which includes negative values. This matters for comparisons. (An `unsigned` would always satisfy $x < -1$, which may be problematic)

- `Exp Field == 0xFF`: Represents $\pm\infty$ or NaN. These clearly fall outside `int` range, so we return `0x80000000` as it is required by the instruction.
- `Exp Field == 0x00`: Represents a denormalized value. It lies strictly between 0 and 1 and rounds to 0, so we return 0.
- Normalized: For $ne < 0$ (magnitude < 1), return 0. For $ne > 30$(exceeds `int` range), return `0x80000000`. Otherwise, compute `re` by first restoring the hidden leading 1 as `((m << 7) | 0x40000000)` and then shifting by `(30 - ne)`. Since the magnitude is nonnegative before applying the sign, use `s` to decide whether to return `re` or `-re`.

Here, the shift count uses 30 bits(excluding the sign bit and the hidden leading 1). Larger `ne` means fewer right shifts and thus a larger integer result.

3. Result

```
[math@nyeoglya datalab2]$ ./dlc bits.c
[math@nyeoglya datalab2]$ ./btest
Score   Rating  Errors  Function
 2      2       0       negate
 3      3       0       isLess
 2      2       0       float_abs
 4      4       0       float_twice
 4      4       0       float_i2f
 4      4       0       float_f2i
Total points: 19/19
[math@nyeoglya datalab2]$
```

The above screenshot shows the output of the grading program `btest` and programming rule checking program `dlc` run on my local device. I obtained 19 points out of 19 without violating any programming rules. The same result can be found on the programming server(programming2.postech.ac.kr).

4. References

[1] Operator Precedence: `https://cppreference.com/w/cpp/language/operator_precedence.html`