University of Colorado - Boulder

CSPB 2400
Computer Systems | Summer 2024

Lab 1: Datalab

Sam Walker

Hoang Truong

Monday, June 3, 2024

College of Engineering & Applied Science
UNIVERSITY OF COLORADO BOULDER

Part 1

Execution Check (40 points)

Code Comment Check (20 points)

GitHub Commit History Check (10 points)

GitHub Repo Link: Repo Link [https://github.com/cu-cspb-2400-summer-2024/lab1-datalab-sawa9885]

Screenshot of driver.pl:

```
jovyan@jupyter-sawa9885:~/CSPB 2400/lab1-datalab-sawa9885$ ./driver.pl
1. Running './dlc -z' to identify coding rules violations.

2. Running './bddcheck/check.pl -g' to determine correctness score.

3. Running './dlc -Z' to identify operator count violations.

4. Running './bddcheck/check.pl -g -r 2' to determine performance score.

5. Running './dlc -e' to get operator count of each function.

Correctness Results     Perf Results
Points  Rating  Errors  Points  Ops     Puzzle
1       1       0       2       4       bitOr
1       1       0       2       1       isZero
1       1       0       2       2       tmax
2       2       0       2       9       anyOddBit
2       2       0       2       7       fitsBits
2       2       0       2       3       leastBitPos
3       3       0       2       10      isAsciiDigit
3       3       0       2       13      isLessOrEqual
3       3       0       2       13      reverseBytes
4       4       0       2       36      bitCount
4       4       0       2       6       logicalNeg
4       4       0       2       23      trueFiveEighths
2       2       0       2       7       float_neg
4       4       0       2       15      float_twice

General Points = 54/54 [30/30 Corr + 24/24 Perf]
Extra Points = 10/10 [6/6 Corr + 4/4 Perf]

Conversion to 40% of DataLab grading (the remaining 60% will come from interview grading)
Execution Grade = 40
Execution Extra = 4
```

Fig. 1 Execution of driver.pl for DataLab

Part 2-i

Rating 2 Question Explanation

Function: fitsBits

High Level:

Check if a given integer can be represented using n bits by shifting bits and determining if any ones are converted to zeroes

Psuedocode:

1) Shift the integer left by (32 - n) bits to remove the unwanted bits.
2) Shift it back to the right by (32 - n) bits to restore the value.
3) Check if the original integer equals the result.

Note: When actually implementing this function it is necessary to replace subtraction with addition of two's complement.

```
1 int fitsBits(int x, int n) {
2 int shift =32- n;
3 return (x <<shift >>shift) ==x;
4 }
```

Fig. 2 Pseudocode Example

Binary Representation:

1) Input: x = 5, n = 3 (binary: 0000 0101)
2) Intermediate: 5 « (32 - 3) = 5 « 29 = 10100000000000000000000000000000
3) Shift back: 10100000000000000000000000000000 » 29 = 00000000000000000000000000000101
4) Output: true (1)

As you can see above, if the input was instead 8, which would be 4 bits long, then it would not be able to fit.

16 Bit vs 32 Bit:

To make this function work for 16 bit inputs, all that would change is the constant 32. For 16 bit, line 2 would be 16-n. And of course in actual implementation we need to use the two's complement representation and add rather than using subtraction.

Part 2-ii

Rating 3 Question Explanation

Function: reverseBytes

High Level:

Reverse the byte order of a given integer by saving each byte and then outputting them in reverse order.

Psuedocode:

1) Extract each byte using bitwise AND and shifts comparing the given bytes to bytes full of ones '0xFF'.
2) Combine the bytes in reversed order using bitwise OR and shifts to construct the full 32 bit output.

```
unsigned int reverseBytes(int x) {
    int byte1 =(x &0xFF);
    int byte2 =(x >>8) &0xFF;
    int byte3 =(x >>16) &0xFF;
    int byte4 =(x >>24) &0xFF;

    return (byte1 <<24) |(byte2 <<16) |
         (byte3 <<8) |byte4;
}
```

Fig. 3 Example

Binary Representation:

1) Input: x = 0x12345678 (binary: 00010010 00110100 01010110 01111000)
2) Extracted bytes: 0x12, 0x34, 0x56, 0x78
3) Reversed bytes: 0x78, 0x56, 0x34, 0x12
4) Output: 0x78563412 (binary: 01111000 01010110 00110100 00010010)

An important distinction is that this functions reverses the byte order and not the bit order. That means that each byte has the same 8 bits in the same order.

16 Bit vs 32 Bit:

To make this function work for 16-bit inputs, there would be 2 bytes and shifting would have to be adjusted accordingly, a shift of 8 for each byte.

Part 2-iii

Rating 4 Question Explanation

Function: trueFiveEighths

High Level:

Multiply a given integer by 5/8, rounding toward zero, using bitwise operations to handle both positive and negative integers correctly, all while avoiding overflow.

Psuedocode:

1) Adjust the integer for rounding if negative by adding 7 before dividing.
2) Divide the adjusted integer by 8.
3) Multiply the result by 5.
4) Handle the remainder separately to avoid overflow and ensure proper rounding. If the original number is negative, then the remainder is the inverse and we convert it to positive before readjusting it back to negative in the last step.
5) Combine the results to get the final value.

```
1  int trueFiveEighths(int x) {
2      int isNegative =x >>31;
3
4      int adjustment =isNegative &7;
5      int adjusted =x +adjustment;
6
7      int div8 =adjusted >>3;
8      int mult5 =(div8 <<2) +div8;
9
10     int remainder =(isNegative &(~x +1)) +(~isNegative &x) &7;
11     int rem5 =(remainder <<2) +remainder;
12     int rem8 =rem5 >>3;
13
14     return mult5 +(isNegative &(~rem8 +1)) +(~isNegative &rem8);
15 }
```

Fig. 4 Example

Note: This code is much higher than the optimal operations for this problem, but this is all I could get working. The long set of operations on line 10 and line 14 are absolute valuing the remainder and then reapplying the sign before adding it with the original result.

Binary Representation:

1) Positive Example:
    1) Input: x = 16 (binary: 00000000000000000000000000010000)
    2) Divide by 8: div8 = 16 / 8 = 2 (binary: 00000000000000000000000000000010)
    3) Multiply by 5: mult5 = 2 * 5 = 10 (binary: 00000000000000000000000000001010)
    4) Output: 10
2) Negative Example:
    1) Input: x = -16 (binary: 11111111111111111111111111110000)
    2) Adjust for rounding: adjusted = -16 + 7 = -9
    3) Divide by 8: div8 = -9 / 8 = -2 (binary: 11111111111111111111111111111110)
    4) Multiply by 5: mult5 = -2 * 5 = -10 (binary: 11111111111111111111111111110110)
    5) Output: -10

16 Bit vs 32 Bit:

To make this function work for another bit size, such as 16 bits, we first need to change the shift amount for checking if the number is negative, i.e. changing it from 31 to 15. Everything else would run the same.

Key Points:

There are several key distinctions that this approach addresses:

1) By dividing by 8 first and then multiplying by 5, we avoid potential overflow issues.
2) Negative numbers require special handling to ensure correct rounding. Adding 7 to the negative value before dividing ensures that the rounding is towards zero.
3) After dividing by 8, the remainder is multiplied by 5 and divided by 8 again to get the correct result. For negative numbers, the remainder must be handled in two's complement form, so it is essential to take the absolute value and convert it back if necessary.