NetID: ms591

Mariam Sulakian

Professor Sorin

CS 250: Homework 1

FROM C TO BINARY

Representing Datatypes in Binary

1. 39{10} = 0010 0111{2}

Calculation	Outcome	Remainder
39		
39/2	19	1
19/2	9	1
9/2	4	1
4/2	2	0
2/2	1	0
1/2	0	1

2. -27{10} = 1110 0101{2}

   Calculate for 27{10} then invert the 0s and 1s and add 1 to it.

Calculation	Outcome	Remainder
27		
27/2	13	1
13/2	6	1
6/2	3	0
3/2	1	1
1/2	0	1

11011 → 00100 → 00100 + 1 = 00101

3. 39.0{10} to 32-bit IEEE floating point: **0x421c000**

The **integral** part is 10 0111. The **fractional** is 0000. Together, 100111.0000.

Normalize: 1.001110000 * 2^5
Sign is +, which = 0.

Exponent:
Add the bias to the exponent of two, and place it in the exponent field. The bias is 2^(k−1) − 1, where k is the number of bits in the exponent field.
For IEEE 32-bit, k = 8, so the bias is 2^(8−1) − 1 = 127.
In this case, 5 + 127 = 132 = 10000100

31 sign bit	30-23 Exponent	22-0 Mantissa
0	10000100	00111000000000000000000

Binary to Hex conversion: 421c000

A: Binary groups of 4	0100 0010 0001 1100 0000 0000 0000 0000							
B: Binary place value	8421 8421 8421 8421 8421 8421 8421 8421							
A*B	0400 0020 0001 8400 0000 0000 0000 0000							
Sum of 4 C's	4	2	1	12	0	0	0	0
Hexadecimal	4	2	1	C	0	0	0	0

Hence, 39.0{10} is **0x421c000.**

4. -1.125{10} to 32-bit IEEE floating point: **0xbf900000**
   The **integral** part is 1. The **fractional** is 001. Together, 1.001.

   0.125*2 = <u>0</u>.25

   0.25*2 = <u>0</u>.5

   0.5*2 = <u>1</u>

   Normalize: 1.001 * 2^0

   Sign is -, which = 1.

   Exponent:

   Add the bias to the exponent of two, and place it in the exponent field. The bias is 2^(*k*−1) − 1, where *k* is the number of bits in the exponent field.
   For IEEE 32-bit, *k* = 8, so the bias is 2^(8−1) − 1 = 127.
   In this case, 0 + 127 = 127 = 1111111

31 sign bit	30-23 Exponent	22-0 Mantissa
1	01111111	00100000000000000000000

Binary to Hex conversion: bf900000

A: Binary groups of 4	1011 1111 1001 0000 0000 0000 0000 0000
B: Binary place value	8421 8421 8421 8421 8421 8421 8421 8421
A*B	8021 8421 8001 0000 0000 0000 0000 0000
Sum of 4 C's	11 15 9 0 0 0 0 0
Hexadecimal	B F 9 0 0 0 0 0

Hence, 1.125{10} i-s **0xbf900000.**

5. "2017: K>Roy" in ASCII: **32 30 31 37 3A 20 4B 3E 52 6F 79 \0**

Char	2	0	1	7	:	SP	K	>	R	o	y
Dec	50	48	49	55	58	32	75	62	81	111	121
Hex	32	30	31	37	3A	20	4B	3E	52	6F	79

Dec to Hex (example): 50{10} = 32{16}

Division	Result	Remainder (in HEX)
50/16	3	2
3/16	0	3

6. 2^35 cannot be represented by a 32-bit computer, which can represent positive numbers up to 2^31-1 and negative numbers down to -2^31. Anything outside this range cannot be represented. If a result exceeds these limits, the number will wrap.

```
float* e_ptr;

int main() {
     float a = 21.34;
     e_ptr = &a;
     float* b_ptr = (float*) malloc (2*sizeof(float));
     b_ptr[0] = 7.0;
     *(b_ptr+1) = 4.0;
     float c = foo(e_ptr, b_ptr, b_ptr[1]);
     free b_ptr;
     if (c > 10.5){
          return 0;
     } else {
          return 1;
     }
}

float foo(float* x, float *y, float z){
     if (*x > *y + z){
          return *x;
     } else {
          return *y+z;
     }
}
```

Memory as an Array of Bytes

7. Where do the variables live?

a	stack
b_ptr	stack
*b_ptr	heap
*e_ptr	stack (* dereferences e_ptr; e_ptr which is a global variable but *e_ptr is a stack variable)
b_ptr[0]	heap

8. **The value returned by main is 0.** We define a pointer to e_ptr as a global variable outside the main. In the main, we set the stack variable a to float value of 21.34. Then we set e_ptr to *point* to a (`e_ptr = &a`). We have a pointer b_ptr of type float to which we allocate a memory of double the size of float (4 bytes). b_ptr is a heap variable. The next line we define the first data point in the b_ptr array to be 7.0. Then, we have *(b_ptr + 1) = 4.0. Here we are assigning b_ptr+1, or b_ptr[1], to the value 4.0 (let the first position in b_ptr point to the value 4.0). We define the float c to be the float represented by the foo with our imputed variables. The first two (e_ptr and b_ptr) we are passing references to. "foo" here returns 21.34 based on calculations in the function. When we free b_ptr, we deallocate the space taken up in the memory for that variable. We arrive at our if statement. Since c is 21.34 and thus greater than 0, main( ) returns 0.

Compiling C Coding

9. Using the optimized code, the runtime is much faster at 3.099 seconds for the user compared to the unoptimized code at 7.329 seconds as shown below. The user time represents the time spent for the CPU within the process and outside the kernel.

   The Real time for the optimized code is 3.133 seconds compared to the 7.379 seconds taken by the unoptimized code. This means that less time is taken for the optimized code versus the unoptimized code to run from start to finish (Real). The Sys time for both, however, is 0.007 seconds. This means that the time spent in the kernel and within the process is the same for both the optimized and unoptimized codes.

```
ms591@login-teer-14 [production] ~ $ g++ -O3 -o optimized  prog.c
ms591@login-teer-14 [production] ~ $ time ./optimized
C[67][993]=1809312841

real    0m3.133s
user    0m3.099s
sys     0m0.007s
ms591@login-teer-14 [production] ~ $ g++ -O0 -o unoptimized prog.c
ms591@login-teer-14 [production] ~ $ time ./unoptimized prog.c
C[67][993]=1809312841

real    0m7.379s
user    0m7.329s
sys     0m0.007s
```

Writing C Coding

10. fibtimes2.c

11. recurse.c

12. HoopStats.c