AERE 361 Lab 9

Jordan Reese

March, 23 2018

Exercise 2

The ranges of the unsigned systems.

- 8-bit = 0 to 255
- 16-bit = 0 to 65, 535
- 32-bit = 0 to 4,294, 967, 295
- 64-bit = 0 to 18, 446, 744, 073, 709, 551, 615

The ranges of the signed systems.

- 8-bit = -128 to 127
- 16-bit = -32,768 to 32,767
- 32-bit = -2,147,483,648 to 2,147,483,647
- 64-bit = -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

The value of the 8-bit unsigned representation

- 88 = 0101 1000
- 0 = 0000 0000
- 1 = 0000 0001
- 127 = 0111 1111
- 255 = 1111 1111

The value of the 2's complement 8-bit representation

- +88 = 0101 1000
- -88 = 1010 1000
- -1 = 1111 1111

- 0 = 0000 0000 and 1000 0000
- 1 = 0000 0001
- -128 = 1000 0000
- 127 = 0111 1111

The value of the 8-bit signed representation

- 88 = 0101 1000
- -88 = 1101 1000
- -1 = 1000 0001
- 0 = 0000 0000
- 1 = 0000 0001
- -127 = 0111 1111
- 127 = 1111 1111

The value of the 1's complement 8-bit representation

- 88 = 0101 1000
- -88 = 1010 0111
- -1 = 1111 1110
- 0 = 0000 0000 and 1111 1111
- 1 = 0000 0001
- -127 = 1000 0000
- 127 = 0111 1111

1 Exercise 3

The largest and smallest normalized positive numbers

- Largest in binary is S=0 E=1111 1110 F=23 ones
  Factoring in the leading point and the bias we have 1.(23 ones) in binary converts to about $1.9999 * 2^{127} = 3.4028235 * 10^{38}$
  It is the same for negative numbers but S = 1.
- Smallest in binary is S = 0 E = 0000 0001 F = 23 zeros
  Factoring in the leading point and the bias we have 1.(23 zeros) in binary converts to about $1.0 * 2^{-126} = 1.17549435 * 10^{-38}$
  It is the same for negative numbers but S = 1.

- Denormalized is the process but our E is only zeros.
  The largest is S = 0 E = 0000 0000 and F = 23 ones The equation is $(1 - 2^{-23}) * 2^{-126} = 1.1754942 * 10^{-38}$ and the negative is the same but S = 1.

- The smallest is S = 0 E = 0000 0000 F = 23 zeros
  The equation is $1 * 2^{-23} * 2^{-126} = 2^{-149} = 1.4 * 10^{-45}$ and the negative is the same but S = 1.

Exercise 4

Minimum value for normalized 64-bit: 2.22507 E-308
Maximum value for normalized 64-bit: 1.797693 E+308
Minimum value for denormalized 64-bit: 4.9 E-324
Maximum value for denormalized 64-bit: 4.4501477 E-308

2 Exercise 5

I want to start by saying this section is very gross. I had to read multiple documents and chat with a professional programmer in order to get this program in working order, and for it to work against most inputs. One of the things I struggle with most in this lab was trying to understand what the lab actually wanted us to do. We needed to create our own data type. I set up struct. This is shown below

```
  char *mantissa;
  char *expo;
  int sign;
  int expobits;
  int mantbits;
};
```

The struct will be passed through functions in order to properly initialize the struct and set values to it. The other thing we had to do for this lab was dynamically allocate memory to our values, so there is a function used to allocate the struct. The last function simply prints the values in my struct. Setting up all of these functions isn't any less typing, but it does make my main code really really pretty. I will list the functions below and talk briefly about them, because I know you understand what they are saying.

Initialize MyFloat:

```
*in, int expobits, int mantbits){
  in->sign = 0;
  in->mantissa = malloc(mantbits * sizeof(char));
  in->expo = malloc(expobits * sizeof(char));
```

```
  in->expobits = expobits;
  in->mantbits = mantbits;
  return 0;
}
```

This is the function to initialize my struct. I set values to zero and allocate the memory here. Also when this function is called we are after call in main for the bits of exponent and mantissa, so I store those in there respective locations as well.

Freeing Memory:

```
int free_myfloat(struct MyFloat *in){
  free(in->mantissa);
  free(in->expo);
  in->mantissa = NULL;
  in->expo = NULL;
  return 0;
}
```

This was really easy to do. All I used was free the allocated memory and set some numbers to null so they don't start doing silly things.

Printing MyFloat:

```
  int i = 0;
  printf("Your floating point bit representation is: %d ",in->sign);
  for (i = 0; i < in->expobits; i++){
    printf("%d", in->expo[i]);
  }
  printf(" ");
  for (i = 0; i < in->mantbits; i++){
    printf("%d", in->mantissa[i]);
  }
  printf("\n");
  return 0;
}
```

All this code did was set up the printing setup for sign, exponent, and mantissa. Printing in each of the fields was also easy because of how I set up my struct. I just looped over and index and printed out the values.

Setting Values in MyFloat (aka Hell):

```
set_myfloat(struct MyFloat *in,int sign, int whole, double dec){
  in->sign = sign;
  char buffer[1000];
  int shift = 0;
```

```
decimaltobinary(whole,dec,buffer);
printf("%s\n", buffer);

char *p;
p = strchr(buffer, '.');
int numleft = p - buffer;
if (1 == numleft){
  shift = 0;
}
else if(0 == numleft) {
  char *one;
  one = strchr(buffer,'1');
  if (NULL == one){
    in->sign = 0;
    return 0;
  }
  else{
    int numzero = one - (buffer + 1);
    shift = -(numzero + 1);
  }
}
else{
  shift = numleft-1;
}
//bias = 2^(expobits-1)-1
int bias = 0;
bias = pow(2,in->expobits-1) - 1;
int exponent = shift + bias;
char expbuff[100];
decimaltobinary(exponent, 0, expbuff);
printf("%s\n", expbuff);

*strchr(expbuff,'.') = 0;
printf("%s\n", expbuff);
int index = 0;
int lead = in->expobits - strlen(expbuff);
if (lead >= 0){
  for(index = 0; index < strlen(expbuff); index++){
    in->expo[index+lead] = ('0' == expbuff[index]) ? 0 : 1;
  }
}
else{
  //too many bits in expbuff than can fit in expo
  char *h = expbuff - lead;
  for (index = 0; index < strlen(h); index++){
    in->expo[index] = ('0' == h[index]) ? 0 : 1;
  }
}
```

```
  printf("%s\n", buffer);
  char *h = strchr(buffer,'1');
  h++;
  index = 0;
  while(index < in->mantbits && *h){
    if(*h == '.'){
      h++;
    }
    else{
      in->mantissa[index] = ('0' == *h) ? 0 : 1;
      index++;
      h++;
    }
  }

  //HI Brian!
  return 0;
}
```

This is where I wanted to hit my head against a desk multiple times. Here we do the meat of the math and storing values. I won't go super into it because it is very long but I will explain my process. The first thing I did was set the sign because that was found in main. I then passed my input number to my binary to decimal converter I wrote for exercise 1. After that came out I made a pointer for the '.' because I wanted to know whether or not I had one from the converter. Once I had one pointed at then I could find out how much I needed to shift it. I need to know the shift because I needed the value to know how to print my mantissa with a leading 1, and I need the nominal value to calculate the exponent. There are a couple of condition checks for the shift, such as what is there isn't a decimal or what happens if only a decimal was input to the converter. Once I found the shift I was able to compute the bias because after all the -127 is only for 32 bit representation. Once I had the bias I could compute the binary for exponent with the converter. Now I could write the binary into my struct. This gave me a headache because of how the user could input a different number of bit than were calculated by the converter. So there has to be a condition for less than or more than enough bits. The same logic goes for mantissa, but it wasn't as bad to write because I can truncate in the mantissa. Last was the mantissa. I created a pointer that pointed at the first one it found. This would be my leading one. I then just stored by counting the pointer up until it reached the end of the character array. That is the gist of the set function.

Main is pretty boring, so boring I won't even copy the code. All it did was ask for the inputs and call the functions needed. The reason I made my code this way was because the exercise statement was very vague. I wasn't sure if it just wanted IEEE representation, so I made the code print with variable floating point representation.

3 Exercise 6

This code in my opinion was a lot easier to write because the algorithm was basically given to us. All we had to do was set up the code so it did math like math is supposed to be done. I found a quadratic calculator online and then adjusted a couple of lines to do what we were required to do. I copied the base for a couple reasons. It came from an open source code, and I have been told programmers are supposed to be lazy. I did read through and understand the code. I made the inputs doubles so it could hold very large inputs. I also set up the if statement that checked to see whether or not we needed to use the second equation provided. This equation was needed when the denominator of the first equation was equal to zero. We also needed to make sure that we didn't have overflow or underflow. I did this by putting in very weird inputs and checked the roots against my calculator. I couldn't find an input that didn't work.

4 Exercise 7

Last but not least. This was a simple series calculator. We have done this before so the premise was almost the same. I studied my Maclaurin series calculated. The first thing I did was check to see if the I got my flag. I used the bit of code from the last lab used to search for things on the command line. I then just did the small loop to compute the series using the eqaution provided in the exercise. There was also a if else statement to do the calculation for the float flag. Now when running the code with the equation we get the wrong answer. The right answer is 6 but we get nearly 100 when using double precision. This is because we are compounding an error after each iteration because the floating points aren't computing correctly.