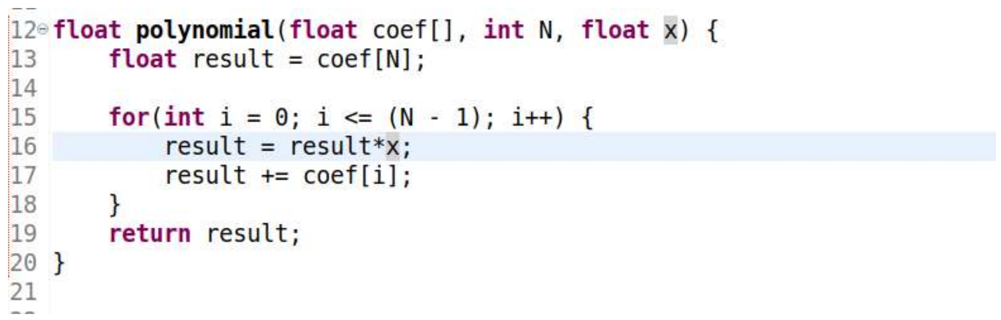
Alisha Patel
I pledge my honor that I have abided by Stevens Honor System.

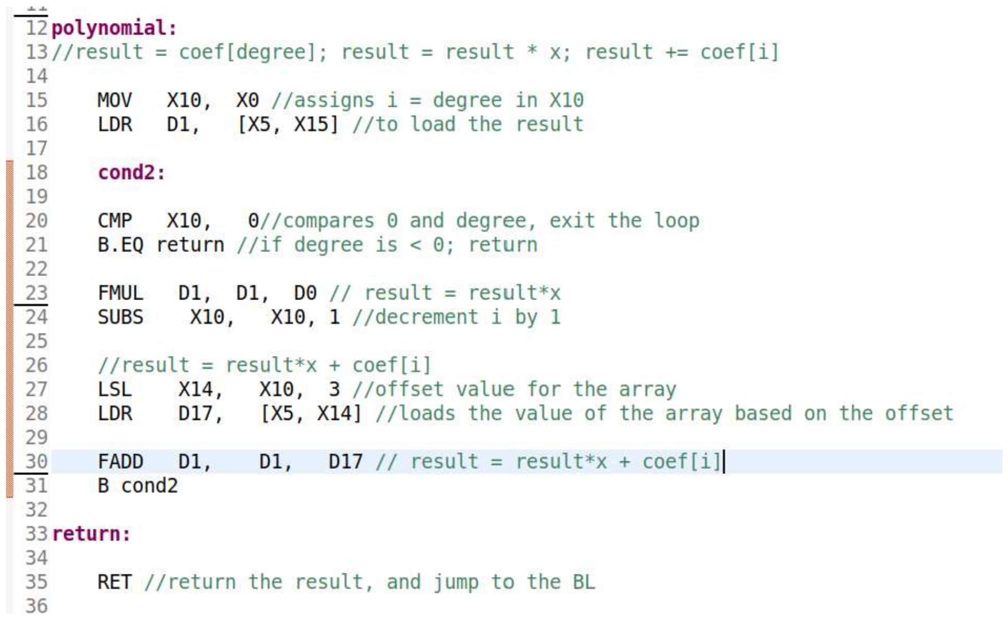
<u>Polynomial:</u>

The polynomial method is used to calculate the function value of the given x. It implements horner's method, as shown below in the c function. The idea is to initialize the result as the coefficient of x^N, which represents the degree. Then, we keep multiplying the result with x and add the next coefficient to result. After iterating through the all the degrees, it returns the result.



```c
float polynomial(float coef[], int N, float x) {
    float result = coef[N];

    for(int i = 0; i <= (N - 1); i++) {
        result = result*x;
        result += coef[i];
    }
    return result;
}

```

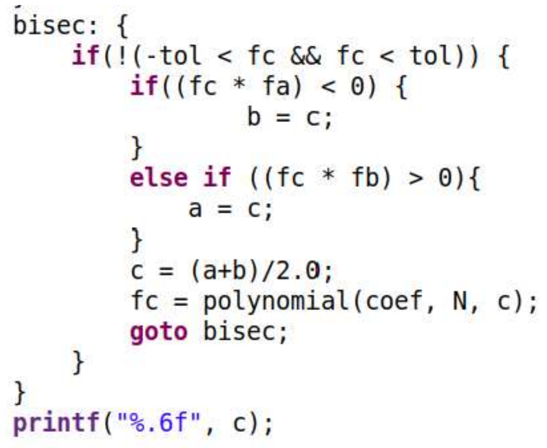
Same logic was followed in the polynomial method when it was written is assembly. The code starts off by moving the degree to X10 for iterating through the equation. D1 holds the calculated value of a polynomial. Since the first part is to multiply with, the last value of the coef[] array should be loaded into D1, in order to achieve the correct placement of the highest/lowest degree of x. Then, it checks if the iterating degree value is 0 (as checking from 0 to (degree - 1) has the same number of iterations as to from degree to 0). If it does equal to 0, it will be terminate the loop, and branch at the return label. The return label, then outputs the D1 as the result. The return label consists of RET, as polynomial method is a procedure call from bisection to evaluate the value of a specific number (in this case, it will be either a, b, c). If the condition is not satisfied, it will then perform the for loop in the c code above. D0 is the specified variable, it will perform a multiplication with the result += result*x. Then, it would decrement the degree by 1, which would be used to calculate the offset value for accessing elements in the coef[]. Since the precision is double, the offset value would be 8. After loading the coef[i], it should be added to the result register, D1. And then unconditional branches back to for loop conditional.



```
polynomial:
//result = coef[degree]; result = result * x; result += coef[i]

    MOV   X10,  X0 //assigns i = degree in X10
    LDR   D1,   [X5, X15] //to load the result

    cond2:

    CMP   X10,   0//compares 0 and degree, exit the loop
    B.EQ return //if degree is < 0; return

    FMUL   D1,  D1,  D0 // result = result*x
    SUBS    X10,   X10, 1 //decrement i by 1

    //result = result*x + coef[i]
    LSL    X14,   X10,  3 //offset value for the array
    LDR    D17,   [X5, X14] //loads the value of the array based on the offset

    FADD   D1,    D1,   D17 // result = result*x + coef[i]
    B cond2

return:

    RET //return the result, and jump to the BL

```

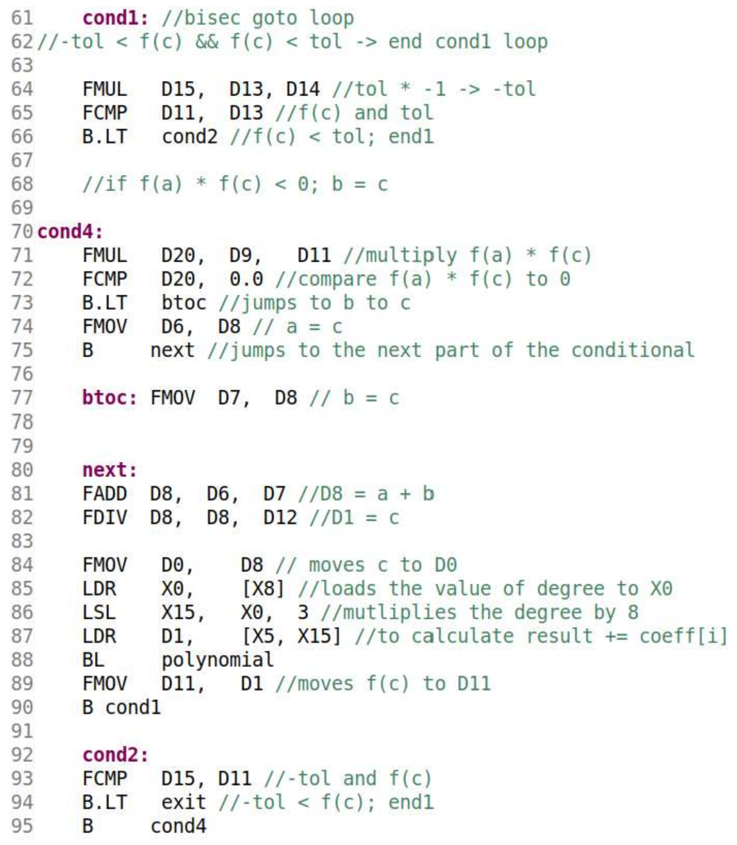
Bisection:

This method will be used to determine the root of the given function. It initially starts off by calculating the mid point, which is this case is c. The leftmost and rightmost bound given will be the user input a and b. If f(c) equals, to 0, then c is the root of the function. When f(c) != 0; there could be two possibilities; f(a) * f(c) </> 0. If it is less than 0, the root lies between a and c, so b becomes the midpoints, then c is calculated, then based on c, f(c) is calculated to check the initial condition again. If f(b)*f(c) < 0, which would imply that the root lies between b and c, so a will becomes the previous midpoint value, and just as before c and f(c) will be calculated again. But, this only takes place when the |f(c)| < tolerance. To make sure of this condition is satisfied to could broken down to -tol < f(c) and f(c) < tol.



```
bisec: {
    if(!(-tol < fc && fc < tol)) {
        if((fc * fa) < 0) {
                b = c;
        }
        else if ((fc * fb) > 0){
            a = c;
        }
        c = (a+b)/2.0;
        fc = polynomial(coef, N, c);
        goto bisec;
    }
}
printf("%.6f", c);
```

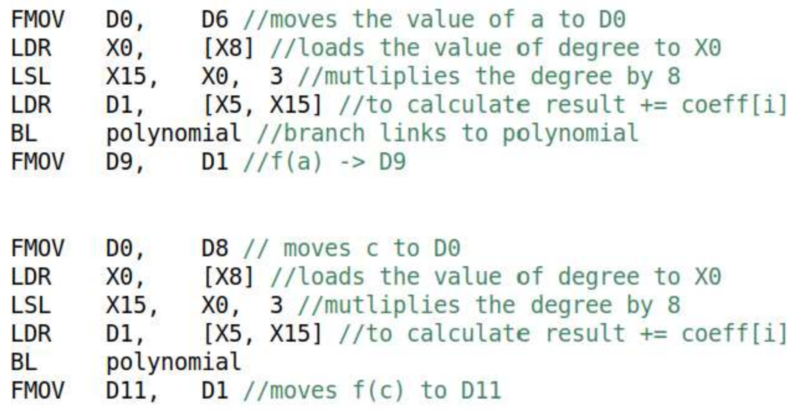
The translated version of the bisection method works the same way. One interesting observation was made during graphing and testing the points is that the f(a) and f(b) would always have opposite signs. So, to make the implementation of the previous c code in assembly, if else statements along with first condition should suffice. The condition 1 here is the if condition in the goto loop. When f(c) < tol, it would jump to the second condition, which asks if -tol < f(c), if this condition is met, it will exit the loop and branch to BL printf statement to terminate the program. If the first condition is not satisfied, it would then continue to the ‘fourth’ condition here which checks whether the product of f(a) and f(c) is less than 0, if it is, then it would jump to the bc label, which sets previous midpoint value of b. If the condition is not met (which is also the same thing as f(b) * f(c) < 0), it would set the previous midpoint value to a. Then, it would proceed to the second part of bisec goto loop; which calculates c again; then f(c) and branches to the first condition. In a case when the first condition is met, but not the second condition, it should loop back to comparing the product of f(a) * f(c) to 0; to avoid infinite loops happening.



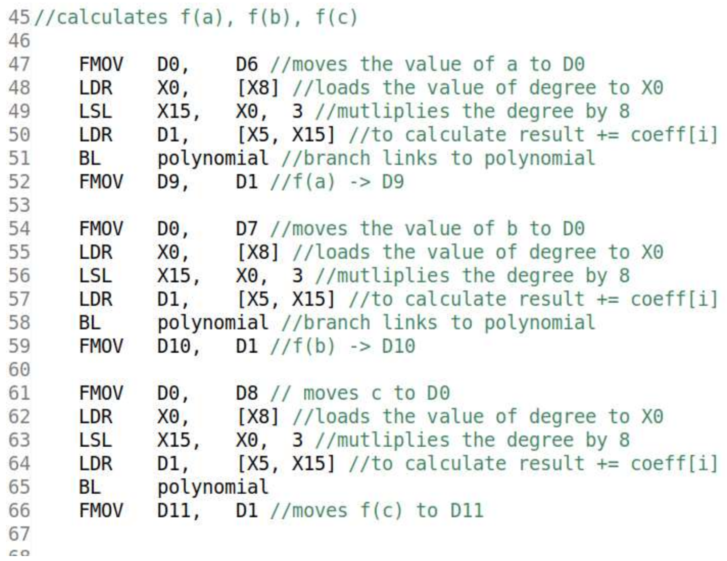
```
    cond1: //bisec goto loop
//-tol < f(c) && f(c) < tol -> end cond1 loop

    FMUL   D15,  D13, D14 //tol * -1 -> -tol
    FCMP   D11,  D13 //f(c) and tol
    B.LT   cond2 //f(c) < tol; end1

    //if f(a) * f(c) < 0; b = c

cond4:
    FMUL   D20,  D9,   D11 //multiply f(a) * f(c)
    FCMP   D20,  0.0 //compare f(a) * f(c) to 0
    B.LT   btoc //jumps to b to c
    FMOV   D6,  D8 // a = c
    B     next //jumps to the next part of the conditional

    btoc: FMOV  D7,  D8 // b = c


    next:
    FADD  D8,  D6,  D7 //D8 = a + b
    FDIV  D8,  D8,  D12 //D1 = c

    FMOV   D0,    D8 // moves c to D0
    LDR    X0,    [X8] //loads the value of degree to X0
    LSL    X15,   X0,  3 //mutliplies the degree by 8
    LDR    D1,    [X5, X15] //to calculate result += coeff[i]
    BL     polynomial
    FMOV   D11,   D1 //moves f(c) to D11
    B cond1

    cond2:
    FCMP   D15, D11 //-tol and f(c)
    B.LT   exit //-tol < f(c); end1
    B     cond4
```

Procedure Calls:

The procedure calls are used to evaluate the values of the function at specific values, a and c. Since there was a entirely different method that calculates function values, a different method that keeps track of the input values of the other method. Procedure calls was necessary part of the program. To calculate the value, D0 was used to input variables (a, b, c), X0 kept the value of degree to iterate through, X15 was calculated offset value to index the last element of coef[] that would be loaded into D1. D1 would be the result (result = result*x + coef[i]). The polynomial function returns the value of D1, then it would stored into their respective registers that keep track of f(a) and f(c). The next instruction that would be read after RET would FMOV (which is right after BL calls). [The f(b) was taken out because it was no longer deemed necessary].



```
FMOV   D0,    D6 //moves the value of a to D0
LDR    X0,    [X8] //loads the value of degree to X0
LSL    X15,   X0,  3 //mutliplies the degree by 8
LDR    D1,    [X5, X15] //to calculate result += coeff[i]
BL     polynomial //branch links to polynomial
FMOV   D9,    D1 //f(a) -> D9


FMOV   D0,    D8 // moves c to D0
LDR    X0,    [X8] //loads the value of degree to X0
LSL    X15,   X0,  3 //mutliplies the degree by 8
LDR    D1,    [X5, X15] //to calculate result += coeff[i]
BL     polynomial
FMOV   D11,   D1 //moves f(c) to D11
```



```
//calculates f(a), f(b), f(c)

    FMOV   D0,    D6 //moves the value of a to D0
    LDR    X0,    [X8] //loads the value of degree to X0
    LSL    X15,   X0,  3 //mutliplies the degree by 8
    LDR    D1,    [X5, X15] //to calculate result += coeff[i]
    BL     polynomial //branch links to polynomial
    FMOV   D9,    D1 //f(a) -> D9

    FMOV   D0,    D7 //moves the value of b to D0
    LDR    X0,    [X8] //loads the value of degree to X0
    LSL    X15,   X0,  3 //mutliplies the degree by 8
    LDR    D1,    [X5, X15] //to calculate result += coeff[i]
    BL     polynomial //branch links to polynomial
    FMOV   D10,   D1 //f(b) -> D10

    FMOV   D0,    D8 // moves c to D0
    LDR    X0,    [X8] //loads the value of degree to X0
    LSL    X15,   X0,  3 //mutliplies the degree by 8
    LDR    D1,    [X5, X15] //to calculate result += coeff[i]
    BL     polynomial
    FMOV   D11,   D1 //moves f(c) to D11
```