Appendix A: Algorithm Implementations in C

A.1 Look Up Table

The following code snippet shows the look up table implementation of RSA encryption/decryption used in this project.

```
#include <stdio.h>
#include <math.h>
#define MAX_TABLE_SIZE 64
int compute_powers_of_two( int exp, int num_powers, int * result) {
    int i = 0;
    int j = 0;
    int temp = 0;

    for(i = 0; i < num_powers; i ++) {
        temp = 1 << i;
        if((temp & exp) != 0) {
            result[j++] = i;
        }
    }
    return j;
}

int ipow(int base, int exp) {
    int result = 1;
    while(exp) {
        if(exp & 1) {
            result *= base;
        }
        exp >>= 1;
        base *= base;
    }
    return result;
}

int compute_lookup_table(int base, int divisor, int * powers, int powers_length,   int * table) {
    int i = 0;
    int length = 0;

    table[length++] = base % divisor;

    for(i = 1; i < 1<<powers[powers_length-1]; i<<=1) {
        table[length] = ipow(table[length-1], 2) % divisor;
        length ++;
    }
    return length;
}

int compute_modulus_from_look_up_table(int * table, int powers_size, int * powers, int divisor) {
    int i = 0;
    int temp = 1;

    temp = table[0];

    for(i=1; i<powers_size; i++) {
        temp *= table[powers[i]];
        temp %= divisor;
    }
    return temp;
}
/*
* This assumes that we need to compute the lookup
* tables for each new call to encrpy.
*/
int rsa_encrypt_lookup_table(int data, int m, int public_key) {
    int powers[MAX_TABLE_SIZE];
    int lookup_table[MAX_TABLE_SIZE];
    int num_powers = compute_powers_of_two(public_key, 32, powers);
    int table_size = compute_lookup_table(data, m, powers, num_powers, lookup_table);
    int x = compute_modulus_from_look_up_table(lookup_table, num_powers, powers, m);
    return x;
}
```

```c
int rsa_decrypt_lookup_table(int data, int m, int private_key) {
    int powers[MAX_TABLE_SIZE];
    int lookup_table[MAX_TABLE_SIZE];
    int num_powers = compute_powers_of_two(private_key, 32, powers);
    int table_size = compute_lookup_table(data, m, powers, num_powers, lookup_table);
    int x = compute_modulus_from_look_up_table(lookup_table, num_powers, powers, m);
    return x;
}

int main(int argc, char * argv[]) {

    int r[MAX_TABLE_SIZE];
    int table[MAX_TABLE_SIZE];

    int exp_value = 2753;
    int base_value = 855;
    int divisor_value = 3233;

    int count = compute_powers_of_two(exp_value, 32, r);
    int table_size = compute_lookup_table(base_value, divisor_value, r, count, table);

    printf("===============RSA TESTS===============\n");
    printf("Inputs: \n");

    int public_key = 17;
    int private_key = 2753;
    int divisor = 3233;
    int plain_text = 123;
    int cipher_text = 0;
    int new_text = 0;

    printf("public_key : %d\n", public_key);
    printf("private_key : %d\n", private_key);
    printf("divisor : %d\n", divisor);
    printf("plain_text : %d\n", plain_text);

    //Test the lookup table method of encryption. This will recompute the
    //lookup table for each call to the encrpyt/decrypt functions.
    printf("\n---LOOKUP TABLE METHOD-----\n");
    cipher_text = rsa_encrypt_lookup_table(plain_text, divisor, public_key);
    new_text = rsa_decrypt_lookup_table(cipher_text, divisor, private_key);
    printf("cipher_text : %d, plain_text : %d\n", cipher_text, new_text);


}
```

A.2 Modular Exponentiation

The following code snippet shows the modular exponentiation algorithm used in this project. Note that all function calls have been removed in this example to make the code as efficient as possible.

```c
#include <stdio.h>
#include <math.h>

int main(int argc, char * argv[]) {
    int result = 1;
    int base = 123;
    int divisor = 3233;
    int exp = 17;

    base = base % divisor;

    while(exp > 0) {
        if(exp % 2 == 1) {
            result = (result * base) % divisor;
        }
        exp >>= 1;
        base = (base * base) % divisor;
    }
    printf("%d\n", result);
}
```

A.3 Montgomery Modular Exponentiation (Unoptimized)

The following code snippet shows unoptimized version of the Montgomery modular exponentiation algorithm used in this project. Note that as many function calls as possible (without affecting the readability of the code) have been removed to decrease the number of branches.

```c
#include <stdio.h>
#include <math.h>

int count_num_bits(int value) {
    int count = 0;

    while(value > 0) {
        count ++;
        value >>= 1;
    }
    return count;
}

int montgomery_multiplication(int x, int y, int m) {
    int t = 0;
    int i = 0;
    int n;
    int iteration_limit = count_num_bits(m);
    int check_bit = 1;

    for(i = 0; i < iteration_limit; i++, check_bit <<= 1) {
        n = (t % 2) + ((x & check_bit) == check_bit) * (y % 2);
        t = ((t + ((x & check_bit) == check_bit) * y + (n * m))) / 2;
    }
    if(t >= m) {
        t = t - m;
    }
    return t;
}

int main(int argc, char * argv[]) {

    int x = 123;
    int m = 3233;
    int e = 17;

    int num_bits = count_num_bits(m);
    int nr = (1 << (2 * num_bits)) % m;
    int z = montgomery_multiplication(1, nr, m);
    int p = montgomery_multiplication(x, nr, m);
    int i = 0;

    for(i = 0; i < num_bits; i++) {
        if(e & (1 << i)) {
            z = montgomery_multiplication(z, p, m);
        }
        p = montgomery_multiplication(p, p, m);
    }
    z = montgomery_multiplication(1, z, m);
    printf("%d\n", z);
}
```

A.4 Montgomery Modular Exponentiation (Optimized)

The following code snippet shows optimized version of the Montgomery modular exponentiation algorithm used in this project. Techniques such as operator strength reduction, initialization/exit condition optimization of loops, local declaration of static variables and suggestion of registers have all been performed. Note that as many function calls as possible (without affecting the readability of the code) have also been removed to decrease the number of branches.

```
#include <stdio.h>
#include <math.h>

long count_num_bits(long value) {
    long count = 0;

    while(value > 0) {
        count ++;
        value >>= 1;
    }
    return count;
}

long long int montgomery_multiplication(
        register long long unsigned int x,
        register long long unsigned int y,
        register long long unsigned int m
){
    int count = 0;
    int n;
    register long long int t = 0;
    long long unsigned int temp = m;
    register long long int check_bit = 1;
    register long long int y_mod = y & 1;
    register long long int x_check;

    // Find the number of bits in the modulus operator
    while(temp > 0){
        count ++;
        temp >>= 1;
    }

    int i = count;

    for(; i != 0; i--, check_bit <<= 1) {
        x_check = ((x & check_bit) != 0);
        n = (t & 1) + x_check * y_mod;
        t = (t + x_check * y + (n * m)) >> 1;
    }
    if(t >= m) {
        t -= m;
    }
    return t;
}

int main(int argc, char * argv[]) {

    long long int x = 123;
    long long int m = 3233;
    long long int e = 17;

    long long int num_bits = count_num_bits(m);
    long long int nr = (1 << (2 * num_bits)) % m;
    long long int z = montgomery_multiplication(1, nr, m);
    long long int p = montgomery_multiplication(x, nr, m);
    long long int i = 0;
    num_bits = 1 << num_bits;

    for(i = 1; i < num_bits; i <<= 1) {
        if(e & i) {
            z = montgomery_multiplication(z, p, m);
        }
        p = montgomery_multiplication(p, p, m);
    }
    z = montgomery_multiplication(1, z, m);
    printf("%lld\n", z);
}
```

Appendix C: Montgomery Multiplication Implementations in VHDL

The following appendix contains the final code used for the VHDL implementation of Montgomery multiplication.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity montgomery_multiplier is
    GENERIC (BITS: integer := 64);
    Port ( clk : in  STD_LOGIC;
           reset : in STD_LOGIC;
           start : in STD_LOGIC;
           ready : out STD_LOGIC;
           multiplier : in  STD_LOGIC_VECTOR (BITS-1 downto 0);
           multiplicand : in  STD_LOGIC_VECTOR (BITS-1 downto 0);
           modulus : in  STD_LOGIC_VECTOR (BITS-1 downto 0);
           product : out  STD_LOGIC_VECTOR (BITS-1 downto 0));
end montgomery_multiplier;

architecture Behavioral of montgomery_multiplier is
    signal setup : STD_LOGIC := '1';
    signal multiplier_r, multiplicand_r, modulus_r, product_c : STD_LOGIC_VECTOR(BITS-1 downto 0) :=
        (others => '0');
    -- We use multiplicand and modulus as constants.
    signal product_sum, mod_temp, sum_temp1, sum_temp2, product_result, mod_test : STD_LOGIC_VECTOR(
        BITS-1 downto 0) := (others => '0');
    signal q_result : STD_LOGIC_VECTOR(1 downto 0) := (others => '0');
begin
    product <= product_c;

    with multiplier_r(0) select
        q_result <= ( product_sum(0) and multiplicand_r(0) ) & ( product_sum(0) xor multiplicand_r(0
            ) ) when '1',
                    '0' & product_sum(0) when others;

    with q_result(1) select
        mod_temp <= modulus_r(BITS-2 downto 0) & '0' when '1',
                    modulus_r when others;

    with multiplier_r(0) select
        sum_temp1 <= product_sum + multiplicand_r when '1',
                     product_sum when others;

    with q_result select
        sum_temp2 <= sum_temp1 when "00",
                     sum_temp1 + mod_temp w hen others;

    product_result <= '0' & sum_temp2(BITS-1 downto 1);
```

```
    ready <= setup;

    process (clk)
    begin
        if (clk'event and clk='1') then
            if setup = '1' then
                if start = '1' then
                    multiplier_r <= multiplier;
                    multiplicand_r <= multiplicand;
                    modulus_r <= modulus;
                    mod_test <= modulus;
                    product_sum <=
                        "0000000000000000000000000000000000000000000000000000000000000000";
                    if not (modulus = (modulus'range => '0')) then
                        setup <= '0';
                    end if;
                end if;
            else
                mod_test <= '0' & mod_test(BITS-1 downto 1);
                multiplier_r <= '0' & multiplier_r(BITS-1 downto 1);
                product_sum <= product_result;
                if mod_test = (mod_test'range => '0') then
                    product_c <= product_sum;
                    setup <= '1';
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```