Assembly code

BigInt.asm

```
void mul_asm(uint64_t a, uint64_t b, uint64_t* hi, uint64_t* lo);
                     ||          ||          ||          ||
                    rcx         rdx         r8          r9
```

- code

```
mul_asm proc
    mov rax, rcx
    mul rdx
    mov [r8], rdx
    mov [r9], rax
    ret
mul_asm endp
```

hi, lo ∈ [0, a) 이어야 함.

```
(9)
uint64_t div_asm(uint64_t hi, uint64_t lo, uint64_t a, uint64_t* r)
          ||              ||          ||          ||          ||
         rax             rcx         rdx         r8          r9
```

```
div_asm proc
    mov rax, rdx
    mov rdx, rcx
    div r8
    mov [r9], rdx
    ret
div_asm endp
```

hi, lo ∈ [0, a) 이어야 함.

```
uint64_t mod_asm(uint64_t hi, uint64_t lo, uint64_t a)
          ||              ||          ||          ||
         rax             rcx         rdx         r8
```

```
mod_asm proc
    mov rax, rdx
    mov rdx, rcx
    div r8
    mov rax, rdx
    ret
mod_asm endp
end
```

BigInt.h

```
#include <stdint.h>
#include <inttypes.h>

class BigInt {
protected:
    uint64_t* data;
    int       size;
public:
    BigInt(int size=1); void resize(int size, bool copy=false);
    BigInt(const uint64_t* data, int size);
    BigInt(const BigInt&);
    void operator=(const BigInt&);
    ~BigInt();

    uint64_t operator[](int i) const { return data[i]; }
    uint64_t& operator[](int i)      { return data[i]; }
    void print(const char* name) const;

    bool operator>=(const BigInt&) const;
    uint64_t mod(uint64_t a) const;
    void add(const BigInt& B, BigInt& C) const;
    void sub(const BigInt& B, BigInt& C) const;
    void mul(uint64_t b, BigInt& C) const;
    void div(uint64_t b, BigInt& Q, uint64_t& r) const;

    double to_real() const;
};
```

unsigend integer

$$\sum_{i=0}^{size-1} data[i]\,\beta^i$$

memory 관련 함수들

접근함수들

C=A+B

C=A-B

C=A*b

A=bQ+r

Comparison & Arithmetic 함수들

BigInt.cpp

```
#include "BigInt.h"

BigInt::BigInt(int size){
    this->size=size; data= new uint64_t[size];
    memset(data, 0, sizeof(uint64_t)*size);
}

BigInt::BigInt(const uint64_t* data, int size){
    this->size=size; this->data=new uint64_t[size];
    memcpy(this->data,data, sizeof(uint64_t)*size);
}

BigInt::BigInt(const BigInt& Z){
    size=Z.size; data= ————
    memcpy(————);
}

void operator=(const BigInt& Z){
    resize(Z.size);

    memcpy(————);
}

void BigInt::resize(int new_size, bool copy){
  if(size!=new_size){
    uint64_t* new_data=new uint64_t[new_size];
    memset(new_data,0,sizeof(uint64_t)*new_size);
    if(copy)memcopy(new_data,data,
        sizeof(uint64_t)*((new_size>size)?size:new_size));
    delete[] data; data=new_data;
    size=new_size;
  }

BigInt::~BigInt(){ delete[] data; }

void BigInt::print(const char* name) const{
    printf("———— %s ——\n", name);
    for(int i=size-1; i>=1; i--)
        printf("%" PRIu64 "*beta^%d+", data[i], i);
    printf("%" PRIu64 "\n", data[0]);
}

bool BigInt::operator>=(const BigInt& B){ const BigInt& A=*this;
    int A_size=A.size, B_size=B.size;
    while(A_size!=B_size){
        if(A_size>B_size) if(A.data[A_size-1]>0) return true; else A_size--;
        else              if(B.data[B_size-1]>0) return false; else B_size--;
    }
    for(int i=A_size-1; i>=0; i--){
        if(A.data[i] > B.data[i]) return true;
        if(B.data[i] > A.data[i]) return false;
    }
    return true;
}
```

```
uint64_t BigInt::mod(uint64_t a) const {
    uint64_t r=0;
    for(int i=size-1; i>=0; i--)
        r = mod_asm(r, data[i] % a, a);
    return r;
}
```

$a[2]+0 \equiv r \pmod a$

$a[1]+\beta\cdot a[2] \equiv a[1]+\beta r \equiv r \pmod a$

$a[0]+a[1]\beta+a[2]\beta^2 \equiv a[0]+\beta\cdot r \equiv r$

```
void BigInt::add(const BigInt& B, BigInt& C) const {
    const BigInt& A = *this;
    int Csize = (A.size > B.size) ? A.size : B.size;
    C.resize(Csize);
    int i=0;
    bool c=0;
    for( ; (i<A.size)&&(i<B.size); i++){
        C[i] = A[i]+B[i]; bool c_next = C[i]<A[i];
        C[i] = C[i]+c ; c = ((C[i]==0) && c) || c_next;
    }
    for(; i<Csize; i++){
        if(i<A.size) C[i] = A[i]+c;
        if(i<B.size) C[i] = B[i]+c;
        c = (C[i]==0) && c;
    }
    if(c){
        C.resize(Csize+1, true);
        C[Csize]=1;
    }
}
```

carry가 발생하면 $C[i]=A[i]+B[i]-\beta < A[i]$

carry propagation

마지막에 carry가 있으면 digit 추가

```
void BigInt::sub(const BigInt& B, BigInt& C) const {
    const BigInt& A = *this;
    assert(A >= B); int Csize = A.size; C.resize(Csize);
    int i=0; bool b=0;
    for( ; i<B.size; i++) {
        C[i] = A[i] - B[i];
        bool b_next = (C[i] > A[i]) ||
                      (b && (C[i]==0));
        C[i] = C[i] - b; b = b_next;
    }
    for( ; i<Csize; i++) { bool b_next = b && (A[i]==0);
        C[i] = A[i] - b;
        b = b_next;
    }
    while((Csize>0) && C[Csize-1]==0)
        Csize--;
    C.resize(Csize, true);
}
```

borrow가 발생하면, C[i] = A[i] - B[i] + β > A[i]

borrow propagation

0 digit 삭제하기.

```
void BigInt::mul(uint64_t b, BigInt& C) const {
    BigInt Cl(size); const BigInt& A = *this;
    BigInt Ch(size+1); Ch[0]=0;
    for(int i=0; i<size; i++)
        mul_asm(A[i], b, &Ch[i+1], &Cl[i]);
    Cl.add(Ch, C);
}
```

```
// A = b·Q + r·

void BigInt::div(uint64_t b, BigInt& Q, uint64_t& r){
    Q.resize(size); r=0;
    const BigInt& A=*this;
    for(int i=size-1; i>=0; i--)
        Q[i]=div_asm(r, A[i], b, &r);
}

//

double BigInt::to_real() const{
    double out=0; double beta=(1ULL<<63); beta*=2;
    for(int i=size-1; i>=0; i--)
        out= out*beta + data[i];
    return out;
}
```