bn_add.c (original)

```
+--  5 lines: Copyright 1995-2018 The OpenSSL Project Authors. All Rights
 * in the file LICENSE in the source distribution or at
 * https://www.openssl.org/source/license.html
 */

#include "internal/cryptlib.h"
#include "bn_local.h"
```

```
/* signed add of b to a. */
int BN_add(BIGNUM *r, const BIGNUM *a, const BIGNUM *b)
{
    int ret, r_neg, cmp_res;

+-- 61 lines: bn_check_top(a);-----------------------------------------
    const BN_ULONG *ap, *bp;
    BN_ULONG *rp, carry, t1, t2;

    bn_check_top(a);
    bn_check_top(b);

    if (a->top < b->top) {
        const BIGNUM *tmp;

        tmp = a;
        a = b;
        b = tmp;
+--  8 lines: }-----------------------------------------------------------
    r->top = max;

    ap = a->d;
    bp = b->d;
    rp = r->d;

    carry = bn_add_words(rp, ap, bp, min);
```

bn_add.c (modified)

```
+--  5 lines: Copyright 1995-2018 The OpenSSL Project Authors. All Rights
 * in the file LICENSE in the source distribution or at
 * https://www.openssl.org/source/license.html
 */

#include "internal/cryptlib.h"
#include "bn_local.h"
#include "bn_par.h"

void *bn_add_sub_words_thread(void *ptr) {
    BN_ULONG c;
    add_sub_args *args = (add_sub_args *) ptr;

    const BN_ULONG* ap = args->a;
    const BN_ULONG* bp = args->b;
    BN_ULONG* rp = args->r;
    BN_ULONG min = args->n;

    if (args->type == '+')
        c = bn_add_words(rp, ap, bp, min);
    else if (args->type == '-')
        c = bn_sub_words(rp, ap, bp, min);

    args->carry = c;
    pthread_exit(NULL);
}

void bn_resolve_carry (BN_ULONG carry, add_sub_args* arg) {
    int i = 0;
    BN_ULONG t;
    while (carry && i < arg->n) {
        t = arg->r[i];
        t = (t + carry) & BN_MASK2;
        carry = (t < carry);
        arg->r[i] = t;
        i++;
    }
    if(i == arg->n) {
        arg->carry += carry;
    }
}
void bn_resolve_borrow (BN_ULONG borrow, add_sub_args* arg) {
    int i = 0;
    BN_ULONG t, t1, c = borrow;
    while (c && i < arg->n) {
        t = arg->r[i];
        t1 = (t - c) & BN_MASK2;
        arg->r[i] = t1;

        //check overflow
        c = (t1 > t);
        i++;
    }
    if(i == arg->n) {
        arg->carry += c;
    }
}
```

```
/* signed add of b to a. */
int BN_add(BIGNUM *r, const BIGNUM *a, const BIGNUM *b)
{
    int ret, r_neg, cmp_res;

+-- 61 lines: bn_check_top(a);-----------------------------------------
    const BN_ULONG *ap, *bp;
    BN_ULONG *rp, carry, t1, t2;

    bn_check_top(a);
    bn_check_top(b);

    // a must be longer than b, if otherwise, swap
    if (a->top < b->top) {
        const BIGNUM *tmp;

        tmp = a;
        a = b;
        b = tmp;
+--  8 lines: }-----------------------------------------------------------
    r->top = max;

    ap = a->d;
    bp = b->d;
    rp = r->d;

    // thread init
    pthread_t thr[NUM_THREADS];
    int rc;

    /* create a thread_data_t argument array */
    add_sub_args thr_data[NUM_THREADS];

    /* create threads, divide array */
    int new_n = min/NUM_THREADS;
    int l_idx = 0;

    for (int i = 0; i < NUM_THREADS; ++i) {
        l_idx = new_n * i;
        // printf("l_idx %d, h_idx %d\n", l_idx, l_idx + new_n);
        thr_data[i].a = &ap[l_idx];
        thr_data[i].b = &bp[l_idx];
        thr_data[i].r = &rp[l_idx];
        thr_data[i].type = '+';

        if (i == (NUM_THREADS - 1))
            thr_data[i].n = new_n + min % NUM_THREADS;
        else
```

```
                                                                thr_data[i].n = new_n;

                                                            if ((rc = pthread_create(&thr[i], NULL, bn_add_sub_words_thread, &
                                                              fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
                                                              return EXIT_FAILURE;
                                                            }
                                                        }
                                                        /* block until all threads complete */
                                                        for (int i = 0; i < NUM_THREADS; ++i) {
                                                            pthread_join(thr[i], NULL);
                                                            // printf("t%d %d\n", i, thr_data[i].carry);
                                                        }

                                                        /* Resolve Carry */
                                                        BN_ULONG tmp_carry;
                                                        for (int i = 0; i < NUM_THREADS - 1; ++i) {
                                                            tmp_carry = thr_data[i].carry;
                                                            bn_resolve_carry(tmp_carry, &thr_data[i+1]);
                                                        }
                                                        carry = thr_data[NUM_THREADS-1].carry;
    rp += min;                                          rp += min;
    ap += min;                                          ap += min;

    while (dif) {                                       while (dif) {
        dif--;                                              dif--;
        t1 = *(ap++);                                       t1 = *(ap++);
+-- 32 lines: t2 = (t1 + carry) & BN_MASK2;-------------+-- 32 lines: t2 = (t1 + carry) & BN_MASK2;-------------
        return 0;                                           return 0;

    ap = a->d;                                          ap = a->d;
    bp = b->d;                                          bp = b->d;
    rp = r->d;                                          rp = r->d;

    borrow = bn_sub_words(rp, ap, bp, min);             // create threads
                                                        pthread_t thr[NUM_THREADS];
                                                        int rc;

                                                        /* create a thread_data_t argument array */
                                                        add_sub_args thr_data[NUM_THREADS];

                                                        /* create threads, divide array */
                                                        int new_n = min/NUM_THREADS;
                                                        int l_idx = 0;

                                                        for (int i = 0; i < NUM_THREADS; ++i) {
                                                            l_idx = new_n * i;
                                                            // printf("l_idx %d, h_idx %d\n", l_idx, l_idx + new_n);
                                                            thr_data[i].a = &ap[l_idx];
                                                            thr_data[i].b = &bp[l_idx];
                                                            thr_data[i].r = &rp[l_idx];
                                                            thr_data[i].type = '-';

                                                            if (i == (NUM_THREADS - 1))
                                                                thr_data[i].n = new_n + min % NUM_THREADS;
                                                            else
                                                                thr_data[i].n = new_n;

                                                            if ((rc = pthread_create(&thr[i], NULL, bn_add_sub_words_thread, &
                                                              fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
                                                              return EXIT_FAILURE;
                                                            }
                                                        }
                                                        /* block until all threads complete */
                                                        for (int i = 0; i < NUM_THREADS; ++i) {
                                                            pthread_join(thr[i], NULL);
                                                            // printf("t%d %d\n", i, thr_data[i].carry);
                                                        }

                                                        /* Resolve Carry */
                                                        BN_ULONG tmp_carry;
                                                        for (int i = 0; i < NUM_THREADS - 1; ++i) {
                                                            tmp_carry = thr_data[i].carry;
                                                            bn_resolve_borrow(tmp_carry, &thr_data[i+1]);
                                                        }
                                                        borrow = thr_data[NUM_THREADS-1].carry;
    ap += min;                                          ap += min;
    rp += min;                                          rp += min;

    while (dif) {                                       while (dif) {
        dif--;                                              dif--;
        t1 = *(ap++);                                       t1 = *(ap++);
+--  8 lines: t2 = (t1 - borrow) & BN_MASK2;------------+--  8 lines: t2 = (t1 - borrow) & BN_MASK2;------------
    r->top = max;                                       r->top = max;
    r->neg = 0;                                         r->neg = 0;
    bn_pollute(r);                                      bn_pollute(r);

    return 1;                                           return 1;
}                                                   }
```

/tmp/yhdaSL_bn_asm.c	crypto/bn/bn_asm.c
+-- 7 lines: Copyright 1995-2016 The OpenSSL Project Authors. All Rights	+-- 7 lines: Copyright 1995-2016 The OpenSSL Project Authors. All Rights
*/	*/
#include <assert.h>	#include <assert.h>
#include <openssl/crypto.h>	#include <openssl/crypto.h>
#include "internal/cryptlib.h"	#include "internal/cryptlib.h"
#include "bn_local.h"	#include "bn_local.h"
	#include "bn_par.h"
#if defined(BN_LLONG) || defined(BN_UMULT_HIGH)	#if defined(BN_LLONG) || defined(BN_UMULT_HIGH)
BN_ULONG bn_mul_add_words(BN_ULONG *rp, const BN_ULONG *ap, int num,	BN_ULONG bn_mul_add_words(BN_ULONG *rp, const BN_ULONG *ap, int num,
BN_ULONG w)	BN_ULONG w)
{	{
	//printf("a\n");
BN_ULONG c1 = 0;	BN_ULONG c1 = 0;
assert(num >= 0);	assert(num >= 0);
if (num <= 0)	if (num <= 0)
return c1;	return c1;
+-- 17 lines: # ifndef OPENSSL_SMALL_FOOTPRINT	+-- 17 lines: # ifndef OPENSSL_SMALL_FOOTPRINT
return c1;	return c1;
}	}
BN_ULONG bn_mul_words(BN_ULONG *rp, const BN_ULONG *ap, int num, BN_ULONG	BN_ULONG bn_mul_words(BN_ULONG *rp, const BN_ULONG *ap, int num, BN_ULONG
{	{
	//printf("ai\n");
BN_ULONG c1 = 0;	BN_ULONG c1 = 0;
assert(num >= 0);	assert(num >= 0);
if (num <= 0)	if (num <= 0)
return c1;	return c1;
+-- 42 lines: # ifndef OPENSSL_SMALL_FOOTPRINT	+-- 42 lines: # ifndef OPENSSL_SMALL_FOOTPRINT
}	}
}	}
#else /* !(defined(BN_LLONG) ||	#else /* !(defined(BN_LLONG) ||
* defined(BN_UMULT_HIGH)) */	* defined(BN_UMULT_HIGH)) */
BN_ULONG bn_mul_add_words(BN_ULONG *rp, const BN_ULONG *ap, int num,	void bn_resolve_carry_mul(BN_ULONG carry, mul_normal_args* arg) {
	int i = 0;
	BN_ULONG t;
	while (carry && i < arg->n) {
	t = arg->r[i];
	t = (t + carry) & BN_MASK2;
	carry = (t < carry);
	arg->r[i] = t;
	i++;
	}
	if(i == arg->n) {
	arg->carry += carry;
	}
	}
	void *bn_mul_add_words_thread(void *ptr)
	{
	mul_normal_args *args = (mul_normal_args *) ptr;
	const BN_ULONG* ap = args->a;
	const BN_ULONG w = args->w;
	BN_ULONG* rp = args->r;
	int num = args->n;
	// printf("num %d\n", num);
	BN_ULONG c = 0;
	BN_ULONG bl, bh;
	bl = LBITS(w);
	bh = HBITS(w);
	# ifndef OPENSSL_SMALL_FOOTPRINT
	while (num & ~3) {
	mul_add(rp[0], ap[0], bl, bh, c);
	mul_add(rp[1], ap[1], bl, bh, c);
	mul_add(rp[2], ap[2], bl, bh, c);
	mul_add(rp[3], ap[3], bl, bh, c);
	ap += 4;
	rp += 4;
	num -= 4;
	}
	# endif
	while (num) {
	mul_add(rp[0], ap[0], bl, bh, c);
	ap++;
	rp++;
	num--;
	}
	args->carry = c;
	pthread_exit(NULL);
	}
	BN_ULONG bn_mul_add_words_par(BN_ULONG *rp, const BN_ULONG *ap, int num,
	BN_ULONG w)
	{
	//printf("b\n");
	BN_ULONG carry = 0;
	assert(num >= 0);
	if (num <= 0)
	return (BN_ULONG)0;
	pthread_t thr[NUM_THREADS];
	int rc;

```
                                BN_ULONG w)
{
    BN_ULONG c = 0;
    BN_ULONG bl, bh;

    assert(num >= 0);
+-- 20 lines: if (num <= 0)----------------------------------------------
        rp++;
        num--;
    }
    return c;
}

BN_ULONG bn_mul_words(BN_ULONG *rp, const BN_ULONG *ap, int num, BN_ULONG
```

```
    /* create a thread_data_t argument array */
    mul_normal_args thr_data[NUM_THREADS];

    /* create threads, divide array */
    int new_n = num/NUM_THREADS;
    int l_idx = 0;
    // printf("%lu\n", w);
    for (int i = 0; i < NUM_THREADS; ++i) {
        l_idx = new_n * i;
        thr_data[i].a = &ap[l_idx];
        thr_data[i].w = w;
        thr_data[i].r = &rp[l_idx];

        if (i == (NUM_THREADS - 1))
            thr_data[i].n = new_n + num % NUM_THREADS;
        else
            thr_data[i].n = new_n;
        // printf("tot %d n %d\n", num, thr_data[i].n);
        if ((rc = pthread_create(&thr[i], NULL, bn_mul_add_words_thread, &
          fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
          exit(EXIT_FAILURE);
        }
    }
    // printf("\n");
    /* block until all threads complete */
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(thr[i], NULL);
        // printf("t%d %d\n", i, thr_data[i].carry);
    }

    /* Resolve Carry */
    BN_ULONG tmp_carry;
    for (int i = 0; i < NUM_THREADS - 1; ++i) {
        tmp_carry = thr_data[i].carry;
        bn_resolve_carry_mul(tmp_carry, &thr_data[i+1]);
    }
    carry = thr_data[NUM_THREADS-1].carry;

    return carry;
}

BN_ULONG bn_mul_add_words_original(BN_ULONG *rp, const BN_ULONG *ap, int n
                                BN_ULONG w)
{
    BN_ULONG c = 0;
    BN_ULONG bl, bh;

    assert(num >= 0);
+-- 20 lines: if (num <= 0)----------------------------------------------
        rp++;
        num--;
    }
    return c;
}

BN_ULONG bn_mul_add_words(BN_ULONG *rp, const BN_ULONG *ap, int num,
                                BN_ULONG w)
{
    if (num > BN_MUL_ADD_NUM_THRESHOLD) {
        return bn_mul_add_words_par(rp, ap, num, w);
    } else {
        return bn_mul_add_words_original(rp, ap, num, w);
    }
}

void *bn_mul_words_thread(void *ptr) {
    mul_normal_args *args = (mul_normal_args *) ptr;

    const BN_ULONG* ap = args->a;
    const BN_ULONG w = args->w;
    BN_ULONG* rp = args->r;
    int num = args->n;

    BN_ULONG bl, bh, carry = 0;

    bl = LBITS(w);
    bh = HBITS(w);
    // printf("w %lx, bh %lx, bl %lx\n", w, bh, bl);

# ifndef OPENSSL_SMALL_FOOTPRINT
    while (num & ~3) {
        mul(rp[0], ap[0], bl, bh, carry);
        mul(rp[1], ap[1], bl, bh, carry);
        mul(rp[2], ap[2], bl, bh, carry);
        mul(rp[3], ap[3], bl, bh, carry);
        ap += 4;
        rp += 4;
        num -= 4;
    }
# endif
    while (num) {
        mul(rp[0], ap[0], bl, bh, carry);
        ap++;
        rp++;
        num--;
    }

    args->carry = carry;
    pthread_exit(NULL);
}

BN_ULONG bn_mul_words_par(BN_ULONG *rp, const BN_ULONG *ap, int num, BN_UL
{
    BN_ULONG carry = 0;

    assert(num >= 0);
    if (num <= 0)
        return (BN_ULONG)0;
```

```
{
    BN_ULONG carry = 0;
    BN_ULONG bl, bh;

    assert(num >= 0);
    if (num <= 0)
+-- 17 lines: return (BN_ULONG)0;----------------------------------------
        mul(rp[0], ap[0], bl, bh, carry);
        ap++;
        rp++;
        num--;
    }
    return carry;
}

void bn_sqr_words(BN_ULONG *r, const BN_ULONG *a, int n)
{
    assert(n >= 0);
    if (n <= 0)
+--868 lines: return;--------------------------------------------------
```

```
    pthread_t thr[NUM_THREADS];
    int rc;

    /* create a thread_data_t argument array */
    mul_normal_args thr_data[NUM_THREADS];

    /* create threads, divide array */
    int new_n = num/NUM_THREADS;
    int l_idx = 0;

    for (int i = 0; i < NUM_THREADS; ++i) {
        l_idx = new_n * i;
        thr_data[i].a = &(ap[l_idx]);
        thr_data[i].w = w;
        thr_data[i].r = &(rp[l_idx]);

        if (i == (NUM_THREADS - 1))
            thr_data[i].n = new_n + num % NUM_THREADS;
        else
            thr_data[i].n = new_n;
        // printf("tot %d l_idx %d, h_idx %d\n", num, l_idx, l_idx + thr_
        if ((rc = pthread_create(&thr[i], NULL, bn_mul_words_thread, &thr_
          fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
          exit(EXIT_FAILURE);
        }
    }
    // printf("\n");
    /* block until all threads complete */
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(thr[i], NULL);
        // printf("t%d %lx\n", i, thr_data[i].carry);
    }

    /* Resolve Carry */
    BN_ULONG tmp_carry;
    for (int i = 0; i < NUM_THREADS - 1; ++i) {
        tmp_carry = thr_data[i].carry;
        bn_resolve_carry_mul(tmp_carry, &thr_data[i+1]);
    }
    carry = thr_data[NUM_THREADS-1].carry;

    return carry;
}
BN_ULONG bn_mul_words_original(BN_ULONG *rp, const BN_ULONG *ap, int num,
{
    BN_ULONG carry = 0;
    BN_ULONG bl, bh;

    assert(num >= 0);
    if (num <= 0)
+-- 17 lines: return (BN_ULONG)0;----------------------------------------
        mul(rp[0], ap[0], bl, bh, carry);
        ap++;
        rp++;
        num--;
    }
    return carry;
}

BN_ULONG bn_mul_words(BN_ULONG *rp, const BN_ULONG *ap, int num, BN_ULONG
{
    if (num > BN_MUL_NUM_THRESHOLD) {
        return bn_mul_words_par(rp, ap, num, w);
    } else {
        return bn_mul_words_original(rp, ap, num, w);
    }
}

void bn_sqr_words(BN_ULONG *r, const BN_ULONG *a, int n)
{
    assert(n >= 0);
    if (n <= 0)
+--868 lines: return;--------------------------------------------------
```

crypto/bn/bn_mul.c	/tmp/FGsasW_bn_mul.c
+-- 4 lines: Copyright 1995-2018 The OpenSSL Project Authors. All Rights	+-- 4 lines: Copyright 1995-2018 The OpenSSL Project Authors. All Rights
* this file except in compliance with the License. You can obtain a copy	* this file except in compliance with the License. You can obtain a copy
* in the file LICENSE in the source distribution or at	* in the file LICENSE in the source distribution or at
* https://www.openssl.org/source/license.html	* https://www.openssl.org/source/license.html
*/	*/
#include <assert.h>	#include <assert.h>
	#include <pthread.h>
#include "internal/cryptlib.h"	#include "internal/cryptlib.h"
#include "bn_local.h"	#include "bn_local.h"
	#include "bn_par.h"
#if defined(OPENSSL_NO_ASM) || !defined(OPENSSL_BN_ASM_PART_WORDS)	#if defined(OPENSSL_NO_ASM) || !defined(OPENSSL_BN_ASM_PART_WORDS)
/*	/*
* Here follows specialised variants of bn_add_words() and bn_sub_words()	* Here follows specialised variants of bn_add_words() and bn_sub_words()
* They have the property performing operations on arrays of different siz	* They have the property performing operations on arrays of different siz
* The sizes of those arrays is expressed through cl, which is the common	* The sizes of those arrays is expressed through cl, which is the common
+--138 lines: * length (basically, min(len(a),len(b))), and dl, which is	+--138 lines: * length (basically, min(len(a),len(b))), and dl, which is
#ifdef BN_RECURSION	#ifdef BN_RECURSION
/*	/*
* Karatsuba recursive multiplication algorithm (cf. Knuth, The Art of	* Karatsuba recursive multiplication algorithm (cf. Knuth, The Art of
* Computer Programming, Vol. 2)	* Computer Programming, Vol. 2)
*/	*/
	pthread_mutex_t thr_count_lock;
	void *bn_mul_recursive_thread(void *ptr) {
	recursive_args *args = (recursive_args *) ptr;
	BN_ULONG *r = args->r;
	BN_ULONG *a = args->a;
	BN_ULONG *b = args->b;
	int n2 = args->n2;
	int dna = args->dna;
	int dnb = args->dnb;
	BN_ULONG *t = args->t;
	int *used_thr = args->used_thr;
	bn_mul_recursive(r, a, b, n2, dna, dnb, t, used_thr);
	pthread_exit(NULL);
	}
	void *bn_mul_part_recursive_thread(void *ptr) {
	recursive_args *args = (recursive_args *) ptr;
	BN_ULONG *r = args->r;
	BN_ULONG *a = args->a;
	BN_ULONG *b = args->b;
	int n = args->n2;
	int tna = args->dna;
	int tnb = args->dnb;
	BN_ULONG *t = args->t;
	int *used_thr = args->used_thr;
	bn_mul_part_recursive(r, a, b, n, tna, tnb, t, used_thr);
	pthread_exit(NULL);
	}
	void start_mul_recursive_thread(pthread_t *thr, recursive_args *arg, BN_UI
	int rc;
	pthread_mutex_lock(&thr_count_lock);
	(*used_thr)++;
	pthread_mutex_unlock(&thr_count_lock);
	set_recursive_arg((*arg), r, a, b, n2, dna, dnb, tmp_thr, used_thr);
	// printf("thread_created %d\n", *(arg->used_thr));
	if ((rc = pthread_create(thr, NULL, bn_mul_recursive_thread, arg))) {
	fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
	exit(EXIT_FAILURE);
	} else {
	// printf("create%d success\n", *used_thr);
	}
	}
	void start_mul_part_recursive_thread(pthread_t *thr, recursive_args *arg,
	int rc;
	pthread_mutex_lock(&thr_count_lock);
	(*used_thr)++;
	pthread_mutex_unlock(&thr_count_lock);
	set_recursive_arg((*arg), r, a, b, n2, dna, dnb, tmp_thr, used_thr);
	// printf("thread_created %d\n", *(arg->used_thr));
	if ((rc = pthread_create(thr, NULL, bn_mul_part_recursive_thread, arg)
	fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
	exit(EXIT_FAILURE);
	} else {
	// printf("create%d success\n", *used_thr);
	}
	}
	int get_used_thread(int* used_thr) {
	pthread_mutex_lock(&thr_count_lock);
	int u = *used_thr;
	pthread_mutex_unlock(&thr_count_lock);
	return u;
	}
/*-	/*-
* r is 2*n2 words in size,	* r is 2*n2 words in size,
* a and b are both n2 words in size.	* a and b are both n2 words in size.
* n2 must be a power of 2.	* n2 must be a power of 2.
* We multiply and return the result.	* We multiply and return the result.
* t must be 2*n2 words in size	* t must be 2*n2 words in size
* We calculate	* We calculate

```c
 * a[0]*b[0]
 * a[0]*b[0]+a[1]*b[1]+(a[0]-a[1])*(b[1]-b[0])
 * a[1]*b[1]
 */
/* dnX may not be positive, but n2/2+dnX has to be */
void bn_mul_recursive(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, int n2,
                      int dna, int dnb, BN_ULONG *t)
{
    int n = n2 / 2, c1, c2;
    int tna = n + dna, tnb = n + dnb;
    unsigned int neg, zero;
    BN_ULONG ln, lo, *p;

+-- 19 lines: # ifdef BN_MUL_COMBA--------------------------------------
        if ((dna + dnb) < 0)
            memset(&r[2 * n2 + dna + dnb], 0,
                   sizeof(BN_ULONG) * -(dna + dnb));
        return;
    }
    /* r=(a[0]-a[1])*(b[1]-b[0]) */
    c1 = bn_cmp_part_words(a, &(a[n]), tna, n - tna);
    c2 = bn_cmp_part_words(&(b[n]), b, tnb, tnb - n);
    zero = neg = 0;
    switch (c1 * 3 + c2) {
    case -4:
        bn_sub_part_words(t, &(a[n]), a, tna, tna - n); /* - */
        bn_sub_part_words(&(t[n]), b, &(b[n]), tnb, n - tnb); /* - */
        break;
    case -3:
        zero = 1;
        break;
    case -2:
        bn_sub_part_words(t, &(a[n]), a, tna, tna - n); /* - */
        bn_sub_part_words(&(t[n]), &(b[n]), b, tnb, tnb - n); /* + */
        neg = 1;
        break;
    case -1:
    case 0:
    case 1:
        zero = 1;
        break;
    case 2:
        bn_sub_part_words(t, a, &(a[n]), tna, n - tna); /* + */
        bn_sub_part_words(&(t[n]), b, &(b[n]), tnb, n - tnb); /* - */
        neg = 1;
        break;
    case 3:
        zero = 1;
        break;
    case 4:
        bn_sub_part_words(t, a, &(a[n]), tna, n - tna);
        bn_sub_part_words(&(t[n]), &(b[n]), b, tnb, tnb - n);
        break;
    }

# ifdef BN_MUL_COMBA
+-- 16 lines: if (n == 4 && dna == 0 && dnb == 0) { XXX: bn_mul_comba4 co

        bn_mul_comba8(r, a, b);
        bn_mul_comba8(&(r[n2]), &(a[n]), &(b[n]));
    } else
# endif                         /* BN_MUL_COMBA */
    {
        p = &(t[n2 * 2]);
        if (!zero)
            bn_mul_recursive(&(t[n2]), t, &(t[n]), n, 0, 0, p);
        else
            memset(&t[n2], 0, sizeof(*t) * n2);
        bn_mul_recursive(r, a, b, n, 0, 0, p);
        bn_mul_recursive(&(r[n2]), &(a[n]), &(b[n]), n, dna, dnb, p);
    }

    /*-
```

```c
 * a[0]*b[0] a_low*b_low
 * a[0]*b[0]+a[1]*b[1]+(a[0]-a[1])*(b[1]-b[0])
 *       a_low*b_low + a_high*b_high + (a_low-a_high)*(b_high-b_low)
 * a[1]*b[1] a_high*b_high
 */
/* dnX may not be positive, but n2/2+dnX has to be */
void bn_mul_recursive(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, int n2,
                      int dna, int dnb, BN_ULONG *t, int *used_thr)
{
    int n = n2 / 2, c1, c2;
    int tna = n + dna, tnb = n + dnb;
    unsigned int neg, zero;
    BN_ULONG ln, lo, *p;

+-- 19 lines: # ifdef BN_MUL_COMBA--------------------------------------
        if ((dna + dnb) < 0)
            memset(&r[2 * n2 + dna + dnb], 0,
                   sizeof(BN_ULONG) * -(dna + dnb));
        return;
    }
    /* r=(a[0]-a[1])*(b[1]-b[0]) */
    c1 = bn_cmp_part_words(a, &(a[n]), tna, n - tna); // a[0] > a[1] ? 1
    c2 = bn_cmp_part_words(&(b[n]), b, tnb, tnb - n); // b[1] > b[0] ? 1
    zero = neg = 0;
    switch (c1 * 3 + c2) {
    case -4: // a[0] < a[1], b[1] < b[0]
        bn_sub_part_words(t, &(a[n]), a, tna, tna - n); /* - */
        bn_sub_part_words(&(t[n]), b, &(b[n]), tnb, n - tnb); /* - */
        break;
    case -3: // a[0] < a[1], b[1] == b[0]
        zero = 1;
        break;
    case -2: // a[0] < a[1], b[1] > b[0]
        bn_sub_part_words(t, &(a[n]), a, tna, tna - n); /* - */
        bn_sub_part_words(&(t[n]), &(b[n]), b, tnb, tnb - n); /* + */
        neg = 1;
        break;
    case -1: // a[0] == a[1], b[1] < b[0]
    case 0: // a[0] == a[1], b[1] =p b[0]
    case 1: // a[0] == a[1], b[1] > b[0]
        zero = 1;
        break;
    case 2: // a[0] > a[1], b[1] < b[0]
        bn_sub_part_words(t, a, &(a[n]), tna, n - tna); /* + */
        bn_sub_part_words(&(t[n]), b, &(b[n]), tnb, n - tnb); /* - */
        neg = 1;
        break;
    case 3: // a[0] > a[1], b[1] == b[0]
        zero = 1;
        break;
    case 4: // a[0] > a[1], b[1] > b[0]
        bn_sub_part_words(t, a, &(a[n]), tna, n - tna);
        bn_sub_part_words(&(t[n]), &(b[n]), b, tnb, tnb - n);
        break;
    }

# ifdef BN_MUL_COMBA
+-- 16 lines: if (n == 4 && dna == 0 && dnb == 0) { XXX: bn_mul_comba4 co

        bn_mul_comba8(r, a, b);
        bn_mul_comba8(&(r[n2]), &(a[n]), &(b[n]));
    } else
# endif                         /* BN_MUL_COMBA */
    {
        pthread_t thr[3];
        recursive_args arg[3];
        int running_cnt = 0, rc;
        BN_ULONG* tp[3];
        p = &(t[n2 * 2]);
        if (!zero) {
            if (get_used_thread(used_thr) < NUM_THREADS) {
                tp[0] = (BN_ULONG *) calloc(n2*2, sizeof(BN_ULONG));
                start_mul_recursive_thread(&(thr[0]), &(arg[0]), &(t[n2])
                running_cnt++;
            } else
                bn_mul_recursive(&(t[n2]), t, &(t[n]), n, 0, 0, p, used_th
        } else
            memset(&t[n2], 0, sizeof(*t) * n2);

        if (get_used_thread(used_thr) < NUM_THREADS) {
            tp[1] = (BN_ULONG *) calloc(n2*2, sizeof(BN_ULONG));
            start_mul_recursive_thread(&(thr[1]), &(arg[1]), r, a, b, n, (
            running_cnt++;
        } else
            bn_mul_recursive(r, a, b, n, 0, 0, p, used_thr);

        if (get_used_thread(used_thr) < NUM_THREADS) {
            tp[2] = (BN_ULONG *) calloc(n2*2, sizeof(BN_ULONG));
            start_mul_recursive_thread(&(thr[2]), &(arg[2]), &(r[n2]), &(a
            running_cnt++;
        } else
            bn_mul_recursive(&(r[n2]), &(a[n]), &(b[n]), n, dna, dnb, p, u

        /* block until all threads complete */
        // printf("running_cnt %d\n", running_cnt);
        for (int i = 0; i < running_cnt; i++) {
            // printf("i %d\n", i);
            if ((rc = pthread_join(thr[i], NULL))) {
                fprintf(stderr, "error: pthread_join, rc: %d\n", rc);
                exit(EXIT_FAILURE);
            } else {
                // printf("join%d success\n", i);
            }
            // printf("t%d %d\n", i, thr_data[i].carry);
            free(tp[i]);
        }
    }

    /*-
```

```
     * t[32] holds (a[0]-a[1])*(b[1]-b[0]), c1 is the sign
     * r[10] holds (a[0]*b[0])
     * r[32] holds (b[1]*b[1])
     */

    c1 = (int)(bn_add_words(t, r, &(r[n2]), n2));

    if (neg) {                  /* if t[32] is negative */
        c1 -= (int)(bn_sub_words(&(t[n2]), t, &(t[n2]), n2));
    } else {
        /* Might have a carry */
        c1 += (int)(bn_add_words(&(t[n2]), &(t[n2]), t, n2));
    }

    /*-
     * t[32] holds (a[0]-a[1])*(b[1]-b[0])+(a[0]*b[0])+(a[1]*b[1])
     * r[10] holds (a[0]*b[0])
     * r[32] holds (b[1]*b[1])
     * c1 holds the carry bits
     */
    c1 += (int)(bn_add_words(&(r[n]), &(r[n]), &(t[n2]), n2));
-----------------------------------------------------------------------
-----------------------------------------------------------------------
    if (c1) {
        p = &(r[n + n2]);
        lo = *p;
        ln = (lo + c1) & BN_MASK2;
        *p = ln;

+-- 14 lines: The overflow will stop before we over write words we should

/*
 * n+tn is the word length t needs to be n*4 is size, as does r
 */
/* tnX may not be negative but less than n */
void bn_mul_part_recursive(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, int n,
                           int tna, int tnb, BN_ULONG *t)
{
    int i, j, n2 = n * 2;
    int c1, c2, neg;
    BN_ULONG ln, lo, *p;

    if (n < 8) {
+-- 45 lines: bn_mul_normal(r, a, n + tna, b, n + tnb);------------------
    if (n == 8) {
        bn_mul_comba8(&(t[n2]), t, &(t[n]));
        bn_mul_comba8(r, a, b);
        bn_mul_normal(&(r[n2]), &(a[n]), tna, &(b[n]), tnb);
        memset(&r[n2 + tna + tnb], 0, sizeof(*r) * (n2 - tna - tnb));
    } else {
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
        p = &(t[n2 * 2]);
        bn_mul_recursive(&(t[n2]), t, &(t[n]), n, 0, 0, p);
        bn_mul_recursive(r, a, b, n, 0, 0, p);
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
        i = n / 2;
        /*
         * If there is only a bottom half to the number, just do it
         */
        if (tna > tnb)
            j = tna - i;
        else
            j = tnb - i;
        if (j == 0) {
            bn_mul_recursive(&(r[n2]), &(a[n]), &(b[n]),
                             i, tna - i, tnb - i, p);
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
            memset(&r[n2 + i * 2], 0, sizeof(*r) * (n2 - i * 2));
        } else if (j > 0) {     /* eg, n == 16, i == 8 and tn == 11 */
            bn_mul_part_recursive(&(r[n2]), &(a[n]), &(b[n]),
                                  i, tna - i, tnb - i, p);
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------------------------------------------------
            memset(&(r[n2 + tna + tnb]), 0,
                   sizeof(BN_ULONG) * (n2 - tna - tnb));
        } else {                /* (j < 0) eg, n == 16, i == 8 and tn == 5

            memset(&r[n2], 0, sizeof(*r) * n2);
            if (tna < BN_MUL_RECURSIVE_SIZE_NORMAL
+--  4 lines: && tnb < BN_MUL_RECURSIVE_SIZE_NORMAL) {------------------
                    i /= 2;
                    /*
                     * these simplified conditions work exclusively becaus
                     * difference between tna and tnb is 1 or 0
                     */
```

```
     * t[n2] holds (a[0]-a[1])*(b[1]-b[0]), c1 is the sign
     * r[0] holds (a[0]*b[0])
     * r[n2] holds (b[1]*b[1])
     */

    c1 = (int)(bn_add_words(t, r, &(r[n2]), n2));

    if (neg) {                  /* if t[n2] is negative */
        c1 -= (int)(bn_sub_words(&(t[n2]), t, &(t[n2]), n2));
    } else {
        /* Might have a carry */
        c1 += (int)(bn_add_words(&(t[n2]), &(t[n2]), t, n2));
    }

    /*-
     * t[n2] holds (a[0]-a[1])*(b[1]-b[0])+(a[0]*b[0])+(a[1]*b[1])
     * r[0] holds (a[0]*b[0])
     * r[n2] holds (b[1]*b[1])
     * c1 holds the carry bits
     */
    c1 += (int)(bn_add_words(&(r[n]), &(r[n]), &(t[n2]), n2));

    // resolve carry on r[n + n2] to last elmt
    if (c1) {
        p = &(r[n + n2]);
        lo = *p;
        ln = (lo + c1) & BN_MASK2;
        *p = ln;

+-- 14 lines: The overflow will stop before we over write words we should

/*
 * n+tn is the word length t needs to be n*4 is size, as does r
 */
/* tnX may not be negative but less than n */
void bn_mul_part_recursive(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, int n,
                           int tna, int tnb, BN_ULONG *t, int *used_thr)
{
    int i, j, n2 = n * 2;
    int c1, c2, neg;
    BN_ULONG ln, lo, *p;

    if (n < 8) {
+-- 45 lines: bn_mul_normal(r, a, n + tna, b, n + tnb);------------------
    if (n == 8) {
        bn_mul_comba8(&(t[n2]), t, &(t[n]));
        bn_mul_comba8(r, a, b);
        bn_mul_normal(&(r[n2]), &(a[n]), tna, &(b[n]), tnb);
        memset(&r[n2 + tna + tnb], 0, sizeof(*r) * (n2 - tna - tnb));
    } else {
        pthread_t thr[3];
        recursive_args arg[3];
        int running_cnt = 0, rc;
        BN_ULONG* tp[3];
        p = &(t[n2 * 2]);

        if (get_used_thread(used_thr) < NUM_THREADS) {
            tp[0] = (BN_ULONG *) calloc(n2*4, sizeof(BN_ULONG));
            start_mul_recursive_thread(&(thr[0]), &(arg[0]), &(t[n2]), t,
            running_cnt++;
        } else
            bn_mul_recursive(&(t[n2]), t, &(t[n]), n, 0, 0, p, used_thr);

        if (get_used_thread(used_thr) < NUM_THREADS) {
            tp[1] = (BN_ULONG *) calloc(n2*4, sizeof(BN_ULONG));
            start_mul_recursive_thread(&(thr[1]), &(arg[1]), r, a, b, n,
            running_cnt++;
        } else
            bn_mul_recursive(r, a, b, n, 0, 0, p, used_thr);

        i = n / 2;
        /*
         * If there is only a bottom half to the number, just do it
         */
        if (tna > tnb)
            j = tna - i;
        else
            j = tnb - i;
        if (j == 0) {
            if (get_used_thread(used_thr) < NUM_THREADS) {
                tp[2] = (BN_ULONG *) calloc(n2*2, sizeof(BN_ULONG));
                start_mul_recursive_thread(&(thr[2]), &(arg[2]), &(r[n2]),
                                           i, tna - i, tnb - i, tp[2], used_thr);
                running_cnt++;
            } else
                bn_mul_recursive(&(r[n2]), &(a[n]), &(b[n]),
                                 i, tna - i, tnb - i, p, used_thr);
            memset(&r[n2 + i * 2], 0, sizeof(*r) * (n2 - i * 2));
        } else if (j > 0) {     /* eg, n == 16, i == 8 and tn == 11 */
            if (get_used_thread(used_thr) < NUM_THREADS) {
                tp[2] = (BN_ULONG *) calloc(n2*2, sizeof(BN_ULONG));
                start_mul_recursive_thread(&(thr[2]), &(arg[2]), &(r[n2]),
                                           i, tna - i, tnb - i, tp[2], used_thr);
                running_cnt++;
            } else
                bn_mul_recursive(&(r[n2]), &(a[n]), &(b[n]),
                                 i, tna - i, tnb - i, p, used_thr);
            memset(&(r[n2 + tna + tnb]), 0,
                   sizeof(BN_ULONG) * (n2 - tna - tnb));
        } else {                /* (j < 0) eg, n == 16, i == 8 and tn == 5

            memset(&r[n2], 0, sizeof(*r) * n2);
            if (tna < BN_MUL_RECURSIVE_SIZE_NORMAL
+--  4 lines: && tnb < BN_MUL_RECURSIVE_SIZE_NORMAL) {------------------
                    i /= 2;
                    /*
                     * these simplified conditions work exclusively becaus
                     * difference between tna and tnb is 1 or 0
                     */
```

```
                    if (i < tna || i < tnb) {
                        bn_mul_part_recursive(&(r[n2]),
                                              &(a[n]), &(b[n]),
                                              i, tna - i, tnb - i, p);
                        break;
                    } else if (i == tna || i == tnb) {
                        bn_mul_recursive(&(r[n2]),
                                         &(a[n]), &(b[n]),
                                         i, tna - i, tnb - i, p);
                        break;
                    }
                }
            }
        }

    /*-
     * t[32] holds (a[0]-a[1])*(b[1]-b[0]), c1 is the sign
     * r[10] holds (a[0]*b[0])
     * r[32] holds (b[1]*b[1])
+-- 41 lines: ------------------------------------------------------------
 * r needs to be n2 words and t needs to be n2*2
 */
void bn_mul_low_recursive(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, int n2,
                          BN_ULONG *t)
{
    int n = n2 / 2;

    bn_mul_recursive(r, a, b, n, 0, 0, &(t[0]));
    if (n >= BN_MUL_LOW_RECURSIVE_SIZE_NORMAL) {
        bn_mul_low_recursive(&(t[0]), &(a[0]), &(b[n]), n, &(t[n2]));
        bn_add_words(&(r[n]), &(r[n]), &(t[0]), n);
        bn_mul_low_recursive(&(t[0]), &(a[n]), &(b[0]), n, &(t[n2]));
        bn_add_words(&(r[n]), &(r[n]), &(t[0]), n);
    } else {
+-- 47 lines: bn_mul_low_normal(&(t[0]), &(a[0]), &(b[n]), n);-----------
            goto err;
    } else
        rr = r;

#if defined(BN_MUL_COMBA) || defined(BN_RECURSION)
    i = al - bl;
#endif
#ifdef BN_MUL_COMBA
    if (i == 0) {
# if 0
        if (al == 4) {
            if (bn_wexpand(rr, 8) == NULL)
                goto err;
            rr->top = 8;
            bn_mul_comba4(rr->d, a->d, b->d);
            goto end;
        }
# endif
        if (al == 8) {
            if (bn_wexpand(rr, 16) == NULL)
                goto err;
            rr->top = 16;
            bn_mul_comba8(rr->d, a->d, b->d);
            goto end;
        }
    }
#endif                          /* BN_MUL_COMBA */
#ifdef BN_RECURSION
    if ((al >= BN_MULL_SIZE_NORMAL) && (bl >= BN_MULL_SIZE_NORMAL)) {
        if (i >= -1 && i <= 1) {
            /*
             * Find out the power of two lower or equal to the longest of
             * two numbers
             */
            if (i >= 0) {
                j = BN_num_bits_word((BN_ULONG)al);
            }
            if (i == -1) {
                j = BN_num_bits_word((BN_ULONG)bl);
            }
            j = 1 << (j - 1);
```

```
                    if (i < tna || i < tnb) {
                        if (get_used_thread(used_thr) < NUM_THREADS) {
                            tp[2] = (BN_ULONG *) calloc(n2*2, sizeof(BN_UI
                            start_mul_part_recursive_thread(&(thr[2]), &(a
                                                  &(a[n]), &(b[n]),
                                                  i, tna - i, tnb - i, tp[2], u
                            running_cnt++;
                        } else
                            bn_mul_part_recursive(&(r[n2]),
                                                  &(a[n]), &(b[n]),
                                                  i, tna - i, tnb - i, p,
                        break;
                    } else if (i == tna || i == tnb) {
                        if (get_used_thread(used_thr) < NUM_THREADS) {
                            tp[2] = (BN_ULONG *) calloc(n2*2, sizeof(BN_UI
                            start_mul_recursive_thread(&(thr[2]), &(arg[2]
                                                  &(a[n]), &(b[n]),
                                                  i, tna - i, tnb - i, tp[2], u
                            running_cnt++;
                        } else
                            bn_mul_recursive(&(r[n2]),
                                             &(a[n]), &(b[n]),
                                             i, tna - i, tnb - i, p, used_
                        break;
                    }
                }
            }
        }

        /* block until all threads complete */
        // printf("running_cnt %d\n", running_cnt);
        for (int i = 0; i < running_cnt; i++) {
            // printf("i %d\n", i);
            if ((rc = pthread_join(thr[i], NULL))) {
                fprintf(stderr, "error: pthread_join, rc: %d\n", rc);
                exit(EXIT_FAILURE);
            } else {
                // printf("join%d success\n", i);
            }
            // printf("t%d %d\n", i, thr_data[i].carry);
            free(tp[i]);
        }
    }

    /*-
     * t[32] holds (a[0]-a[1])*(b[1]-b[0]), c1 is the sign
     * r[10] holds (a[0]*b[0])
     * r[32] holds (b[1]*b[1])
+-- 41 lines: ------------------------------------------------------------
 * r needs to be n2 words and t needs to be n2*2
 */
void bn_mul_low_recursive(BN_ULONG *r, BN_ULONG *a, BN_ULONG *b, int n2,
                          BN_ULONG *t)
{
    int n = n2 / 2;
    int u = 99;
    bn_mul_recursive(r, a, b, n, 0, 0, &(t[0]), &u);
    if (n >= BN_MUL_LOW_RECURSIVE_SIZE_NORMAL) {
        bn_mul_low_recursive(&(t[0]), &(a[0]), &(b[n]), n, &(t[n2]));
        bn_add_words(&(r[n]), &(r[n]), &(t[0]), n);
        bn_mul_low_recursive(&(t[0]), &(a[n]), &(b[0]), n, &(t[n2]));
        bn_add_words(&(r[n]), &(r[n]), &(t[0]), n);
    } else {
+-- 47 lines: bn_mul_low_normal(&(t[0]), &(a[0]), &(b[n]), n);-----------
            goto err;
    } else
        rr = r;

#if defined(BN_MUL_COMBA) || defined(BN_RECURSION)
    i = al - bl;
    // printf("i %d, al %d, bl %d\n", i, al, bl);
#endif
#ifdef BN_MUL_COMBA
    if (i == 0) {
# if 0
        if (al == 4) {
            if (bn_wexpand(rr, 8) == NULL)
                goto err;
            rr->top = 8;
            bn_mul_comba4(rr->d, a->d, b->d);
            goto end;
        }
# endif
    // printf("comba\n");
        if (al == 8) {
            if (bn_wexpand(rr, 16) == NULL)
                goto err;
            rr->top = 16;
            bn_mul_comba8(rr->d, a->d, b->d);
            goto end;
        }
    }
#endif                          /* BN_MUL_COMBA */
#ifdef BN_RECURSION
    if ((al >= BN_MULL_SIZE_NORMAL) && (bl >= BN_MULL_SIZE_NORMAL)) {
        if (i >= -1 && i <= 1) {
            // printf("recursion\n");
            /*
             * Find out the power of two lower or equal to the longest of
             * two numbers
             */
            if (i >= 0) {
                j = BN_num_bits_word((BN_ULONG)al);
            }
            if (i == -1) {
                j = BN_num_bits_word((BN_ULONG)bl);
            }
            j = 1 << (j - 1);
            // printf("j %d\n", j);
```

```
            assert(j <= al || j <= bl);
            k = j + j;
            t = BN_CTX_get(ctx);
            if (t == NULL)
                goto err;
            if (al > j || bl > j) {
--------------------------------------------------------------------------
                if (bn_wexpand(t, k * 4) == NULL)
                    goto err;
                if (bn_wexpand(rr, k * 4) == NULL)
                    goto err;
                bn_mul_part_recursive(rr->d, a->d, b->d,
                                      j, al - j, bl - j, t->d);
            } else {            /* al <= j || bl <= j */

--------------------------------------------------------------------------
--------------------------------------------------------------------------
--------------------------------------------------------------------------
--------------------------------------------------------------------------
--------------------------------------------------------------------------
                if (bn_wexpand(t, k * 2) == NULL)
                    goto err;
                if (bn_wexpand(rr, k * 2) == NULL)
                    goto err;
                bn_mul_recursive(rr->d, a->d, b->d, j, al - j, bl - j, t->
--------------------------------------------------------------------------
            }
            rr->top = top;
            goto end;
        }
    }
#endif                          /* BN_RECURSION */
    if (bn_wexpand(rr, top) == NULL)
        goto err;
    rr->top = top;
--------------------------------------------------------------------------
    bn_mul_normal(rr->d, a->d, al, b->d, bl);

#if defined(BN_MUL_COMBA) || defined(BN_RECURSION)
 end:
#endif
    rr->neg = a->neg ^ b->neg;
+--  9 lines: rr->flags |= BN_FLG_FIXED_TOP;-----------------------------
}

void bn_mul_normal(BN_ULONG *r, BN_ULONG *a, int na, BN_ULONG *b, int nb)
{
    BN_ULONG *rr;

--------------------------------------------------------------------------
    if (na < nb) {
        int itmp;
        BN_ULONG *ltmp;

        itmp = na;
        na = nb;
+--  53 lines: nb = itmp;-------------------------------------------------
```

```
            assert(j <= al || j <= bl);
            k = j + j;
            t = BN_CTX_get(ctx);
            if (t == NULL)
                goto err;
            if (al > j || bl > j) {
                // printf("mul-part-rec\n");
                if (bn_wexpand(t, k * 4) == NULL)
                    goto err;
                if (bn_wexpand(rr, k * 4) == NULL)
                    goto err;
--------------------------------------------------------------------------
--------------------------------------------------------------------------
--------------------------------------------------------------------------

                int used_thread = 1;
                bn_mul_part_recursive(rr->d, a->d, b->d,
                                      j, al - j, bl - j, t->d, &used_threa
            } else {            /* al <= j && bl <= j */
                // al or bl is exacly the power of two
                if (bn_wexpand(t, k * 2) == NULL)
                    goto err;
                if (bn_wexpand(rr, k * 2) == NULL)
                    goto err;
                int used_thread = 1;
                bn_mul_recursive(rr->d, a->d, b->d, j, al - j, bl - j, t->
            }
            rr->top = top;
            goto end;
        }
    }
#endif                          /* BN_RECURSION */
    if (bn_wexpand(rr, top) == NULL)
        goto err;
    rr->top = top;
    // printf("normal\n");
    bn_mul_normal(rr->d, a->d, al, b->d, bl);

#if defined(BN_MUL_COMBA) || defined(BN_RECURSION)
 end:
#endif
    rr->neg = a->neg ^ b->neg;
+--  9 lines: rr->flags |= BN_FLG_FIXED_TOP;-----------------------------
}

void bn_mul_normal(BN_ULONG *r, BN_ULONG *a, int na, BN_ULONG *b, int nb)
{
    BN_ULONG *rr;

    // a must be longer than b, switch if otherwise
    if (na < nb) {
        int itmp;
        BN_ULONG *ltmp;

        itmp = na;
        na = nb;
+--  53 lines: nb = itmp;-------------------------------------------------
```