TECHNISCHE HOCHSCHULE DEGGENDORF

Selected Topics of Embedded Software Development 2

WS-2021/22

Prof. Dr. Martin Schramm

Testing and generating Prime Numbers and Safe Primes using CryptoCore

Group 2 – Team 4
Rashed Al-Lahaseh – 00821573
Vikas Gunti - 12100861

Supriya Kajla – 12100592

Srijith Krishnan – 22107597

Wannakuwa Nadeesh – 22109097

Core RAM Structure

The RAM of the core must be capable of holding all the necessary operands and intermediate values required during the execution of cryptographic algorithms.

The basic structure of the described RAM is figured bellow.

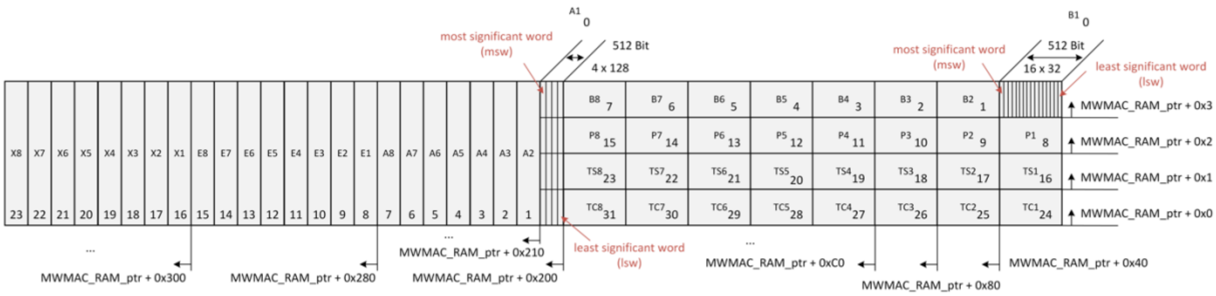


Figure 1: Montgomery Multiplication Core RAM Organization

It features four symbolic horizontal RAM operand locations with MAX_PRERCISION_WIDTH bit each which are organized as eight pieces of MAX_PRERCISION_WIDTH|8 bit each.

The location named B is intended to hold operand B in Montgomery Multiplication and Montgomery Exponentiation operations.

The location named P is intended to hold the modulus.

The location TS usually holds the temporary sum value during Montgomery Multiplications and Montgomery Exponentiation or the first operand in modular addition or subtraction operations.

The location TC usually holds the temporary carry stream during Montgomery Multiplications and Montgomery Exponentiation or the second operand in modular addition or subtraction operations.

Besides the horizontal RAM operand locations three symbolic vertical RAM operand locations with MAX_PRECISION_WIDTH bit each have been defined which are organized as eight pieces of MAX_PRECISION_WIDTH|8 bit each.

The locations named A, E and X for convenience usually are used to hold operand A in Montgomery Multiplication and Montgomery Exponentiation operations as well as the exponent operand E and the auxiliary operand X in Montgomery Exponentiation operations. In addition, all RAM slots are intended to hold intermediate values during the execution of cryptographic algorithms.

Dive into the /driver file

Part 1

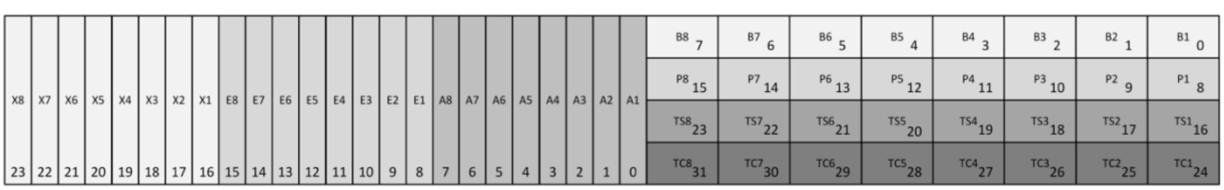
```c
// Add further CryptoCore Functions here...
static void MWMAC_MRT_Prime(MRT_prime_params_t *MRT_prime_params_ptr)
{
    // Commands variables
    u32 mwmac_cmd = 0;
     u32 mwmac_sec_calc = MRT_prime_params_ptr->sec_calc; // Secure calculation bit
     u32 mwmac_precision; // In prime operations precision is standard
    u32 mwmac_f_sel = 1;
     // Temporary counter
     u32 i;
    u32 j;
    u32 k;
    // Temporary random value
    u32 a;
    // Array to hold the random number generated for testing (based on the given precision)
     u32 n[128];
    // Parameter / Temporary arrays
     u32 d[128];
    u32 r[128];
    u32 prec = MRT_prime_params_ptr->prec;
    u32 NSubR[128];
    u32 temporary[128];
     // Word count
     u32 word_count = mwmac_precision/32;
     // Index of last word (least significant word)
     u32 last_word = word_count-1;
     // Flag to check if least sig block is equal to 2
     u32 last_word_is_2;
     // S parameter
     u32 s = 0;
     // Flag to check if other blocks are equal to 0
     u32 blocks;

    // Read the value from PRIME_PRECISIONS struct
    for (i = 0; i < 13; i++) {
        if (PRIME_PRECISIONS[i][0] == prec) {
            mwmac_precision = PRIME_PRECISIONS[i][1];
        }
    }

     // Clear RAM
     Clear_MWMAC_RAM();

     // Copy input number into n and d
     for(i=0; i<word_count; i++) {
         n[i] = MRT_prime_params_ptr->n[i];
         d[i] = MRT_prime_params_ptr->n[i];
     }
```

We start by initializing the important variables needed, which is going to be used within the function

Since the passed value of precision is passed as an integer from the application side, we convert it into the struct

Fill d parameter value, and the input number need to be tested

Coming from an empty or rather unknown memory content.



Part 2

```c
// ---------------------------------------------------------------------------
// Preliminary tests on the number [(n == 2) then prime or (n % 2 == 0) then composite]
// ---------------------------------------------------------------------------

// If number is equal to 2 then it is a prime (check least sig block to be 2)
last_word_is_2 = 0;
if(n[last_word]==2) {
    last_word_is_2 = 1;
} else {
    last_word_is_2 = 0;
}

// Check other blocks to be 0
blocks = 1;
for(i=0; i<last_word; i++) {
    if(n[i]!=0) {
        blocks = 0;
        break;
    }
}

// Make sure that least significant bit is 1 (final check)
if(last_word_is_2==1 && blocks==1) {
    MRT_prime_params_ptr->probably_prime = 1;
    return;
}

// If number is even then it is composite
if(!(n[last_word] & 0x00000001)) {
    MRT_prime_params_ptr->probably_prime = 0;
    return;
}
```

If the generated number is equal to 2 then it is a prime number and if it’s even then defiantly a composite number.

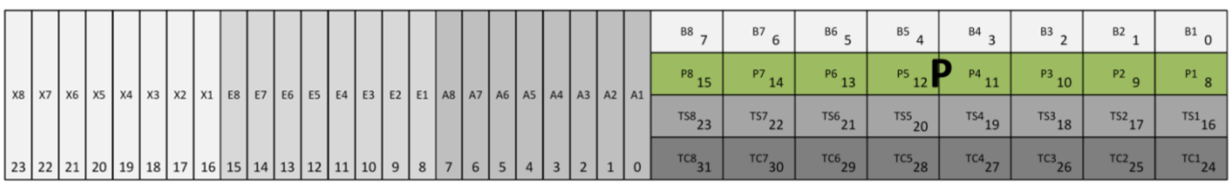
After initializing the d array value with input (n) values we need to confirm the least significant bit and most significant bit by shifting operation and calculate the number of shifts needed.

```c
// ---------------------------------------------------------------------------
// Calculate value of d and s
// ---------------------------------------------------------------------------

// Do minus 1 operation:
//  Since we know received random number is odd we can confidently only AND it
//  with 0xFFFFFFFE to make the right most digit 0 which means number-1
d[last_word] &= 0xFFFFFFFE;

// Shift operation:
// While(least significant block %2==0)
while(!(d[last_word] & 0x00000001)) {
    s++; // Counting the number of shifting
    d[last_word] >>= 1; // Divide by 2 (shift least significant block)
    // Shift other blocks starting from one block after the least significant block
    // [B0 B1 B2 B3 <- B4]   [most sig, ..., least sig]
    for(i=1; i<word_count; i++) {
        // First check if the right most bit of current block is 1.
        // If yes, before shifting, we insert a 1 into the previous (less significant) block from its left side
        if(d[last_word-i] & 0x00000001) // If(d%2==1)   (if right most bit of current part is 1)
            d[last_word-(i-1)] |= 0x80000000; // Sets most left bit of previous block to 1
        d[last_word-i] >>= 1; // Divides current block by 2
    }
}
```

Part 3

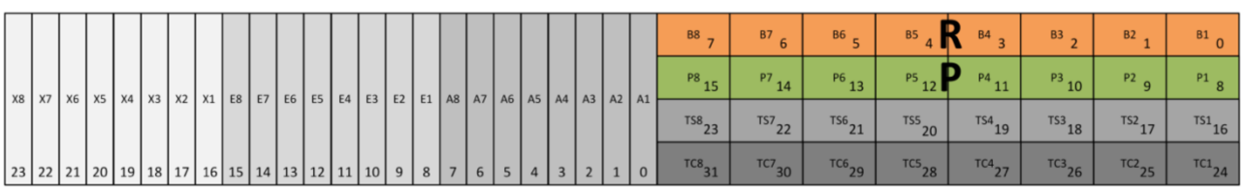
```
 // -----------------------------------------------------------------------------
 // n - R and R [MWMAC_ModR]
 // -----------------------------------------------------------------------------

// Write n to P
for(i=0; i<word_count; i++) {
    iowrite32(n[word_count-1-i], (MWMAC_RAM_ptr+0x2+i*0x4));
 }

// MontR(P, B)
//              Start     Abort       f_sel     sec_calc        precision         operation
 mwmac_cmd = (1 << 0) | (0 << 1) | (mwmac_f_sel << 2) | (mwmac_sec_calc << 3) | (mwmac_precision << 4) | (MONTR << 8)
 //          src_addr                dest_addr            src_addr_e      src_addr_x
        | (MWMAC_RAM_P1 << 12) | (MWMAC_RAM_B1 << 17) | (0x0 << 22) |   (0x0 << 27);
 iowrite32(mwmac_cmd, MWMAC_CMD_ptr);

// Wait until CryptoCore is busy
 while(!mwmac_irq_var);
 mwmac_irq_var = 0;

// Copying value result 'R' from B register to r[128];
for (i = 0; i < word_count; ++i) {
   r[last_word - i] = ioread32(MWMAC_RAM_ptr + 0x3 + i * 0x4);
}
```

So, first the value of P will be written to the appropriate memory location



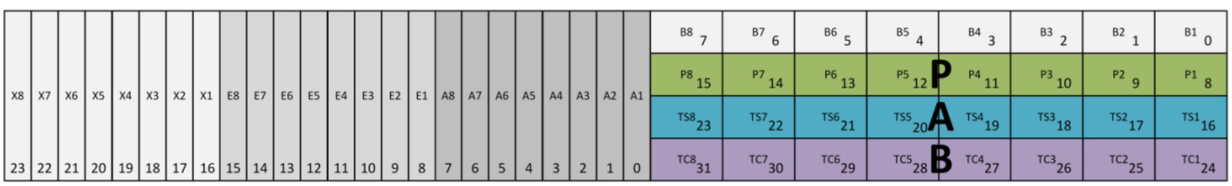
Then the defined starting offsets of the addressed memory locations are used. The encoded value of the precision width has been determined prior.

After successful execution following memory content results. The outcome **R** can be read via device driver from memory, transferred to the application and shown on command line.



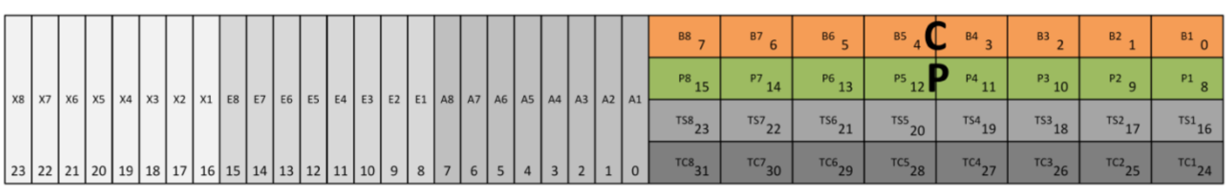
```
// Copying value result 'R' from B register to r[128];
for (i = 0; i < word_count; ++i) {
    r[last_word - i] = ioread32(MWMAC_RAM_ptr + 0x3 + i * 0x4);
}

// Write value of 'N' from n[128] to TS register
 for(i=0; i<word_count; i++) {
     iowrite32(n[word_count-1-i], (MWMAC_RAM_ptr+0x1+i*0x4));
 }

// Write value of 'R' from r[128] to TC register
 for(i=0; i<word_count; i++) {
     iowrite32(r[word_count-1-i], (MWMAC_RAM_ptr+0x0+i*0x4));
 }

// Write value of 'N' from n[128] to P register
 for(i=0; i<word_count; i++) {
     iowrite32(n[word_count-1-i], (MWMAC_RAM_ptr+0x2+i*0x4));
 }

 // ModSub(TS, TC)
 //            Start     Abort       f_sel     sec_calc        precision              operation
 mwmac_cmd = (1 << 0) | (0 << 1) | (mwmac_f_sel << 2) | (mwmac_sec_calc << 3) | (mwmac_precision << 4) | (MODSUB << 8)
 //          src_addr                dest_addr    src_addr_e   src_addr_x
         | (MWMAC_RAM_TS1 << 12) | (0x0 << 17) | (0x0 << 22) | (0x0 << 27);
 iowrite32(mwmac_cmd, MWMAC_CMD_ptr);

 //wait until CryptoCore is busy
 while(!mwmac_irq_var);
 mwmac_irq_var = 0;

 //Copy B to u32 NSubR[128]
 for(i=0; i<word_count; i++) {
     NSubR[word_count-1-i] = ioread32(MWMAC_RAM_ptr+0x3+i*0x4);
 }
```

First the values A, B and P will be written to the appropriate memory locations.



Then the defined starting offsets of the addressed memory locations are used. The encoded value of the precision width has been determined prior.

After successful execution following memory content results. Note that the addressed TS and TC memory locations will be automatically set back to zero. The outcome **C** can be read via device driver from memory, transferred to the application and shown on command line.

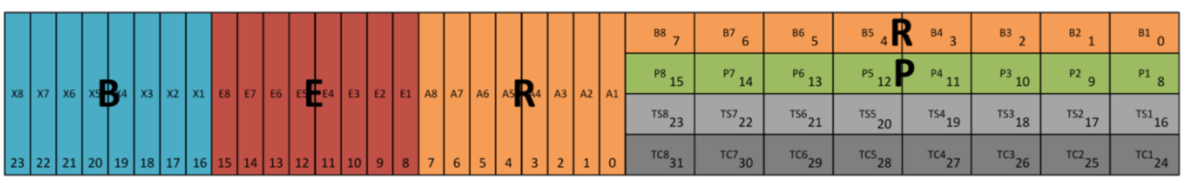


Part 4

```
// ---------------------------------------------------------------------------
// Main Loop
// ---------------------------------------------------------------------------
for(j=0; j < MRT_prime_params_ptr->k; j++) {
    // ===================== a = random.randrange(2, n-2) ====================

    for(i=0; i<word_count; i++) {
        //read a random 32 bit number from TRNG
        a = ioread32(TRNG_FIFO_ptr);

        // Make sure that the final random number will be in the range [2,n-2]. Bit manipulation: 0x0rrrrrrr ... 0xrrrrrr1r
        if(i==0) // Make msb 0 (to make sure our random is <= n-2)
            a &= 0x7FFFFFFF;
        if(i==word_count-1) //make second lsb 1 (to make sure our random is >= 2)
            a |= 0x00000002;

        // Write the random 32 bit number into the appropriate location in X memory
        iowrite32(a, (MWMAC_RAM_ptr+0x300+i));
    }
    // Write value of 'D' from d[128] to E register
    for(i=0; i<word_count; i++) {
        iowrite32(d[word_count-1-i], (MWMAC_RAM_ptr+0x280+i));
    }
    // Write value of 'R' from r[128] to B register
    for(i=0; i<word_count; i++) {
        iowrite32(r[word_count-1-i], (MWMAC_RAM_ptr+0x3+i*0x4));
    }
    // Write value of 'R' from r[128] to A register
    for(i=0; i<word_count; i++) {
        iowrite32(r[word_count-1-i], (MWMAC_RAM_ptr+0x200+i));
    }
```

The main loop will run (k) times where the first task in the main for loop is calculating Montgomery exponentiation. We should copy the random number to the X, (d) to the E, R to B and R to A. Now the board is ready to calculate MontExp(A, B, E, X, P).

At first the values B, E and P will be written to the appropriate memory locations.

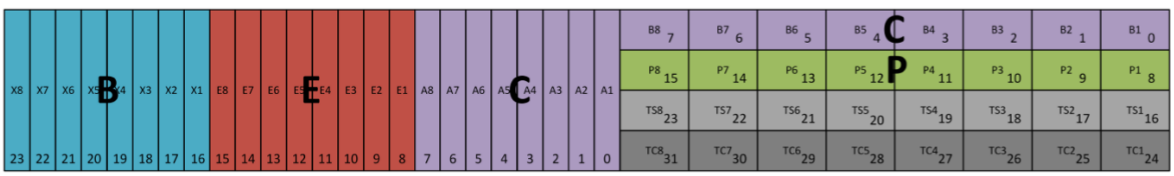


```c
// ============= temporary = Prime_MontExp(a,d,n,prec,r,rinv) ==============

  // MontExp(A, B, E, X, P)
  //            Start     Abort       f_sel     sec_calc        precision          operation
  mwmac_cmd = (1 << 0) | (0 << 1) | (mwmac_f_sel << 2) | (mwmac_sec_calc << 3) | (mwmac_precision << 4) | (MONTEXP << 8)
  //          src_addr                dest_addr           src_addr_e              src_addr_x
            | (MWMAC_RAM_B1 << 12) | (MWMAC_RAM_A1 << 17) | (MWMAC_RAM_E1 << 22) | (MWMAC_RAM_X1 << 27);
  iowrite32(mwmac_cmd, MWMAC_CMD_ptr);
// Wait until CryptoCore is busy
  while(!mwmac_irq_var);
  mwmac_irq_var = 0;

// Copying value result 'temporary' from B register to temporary[128];
  for(i=0; i<word_count; i++) {
      temporary[word_count-1-i] = ioread32(MWMAC_RAM_ptr+0x3+i*0x4);
  }
```

Then the defined starting offsets of the addressed memory locations are used. The encoded value of the precision width has been determined prior.

After successful execution following memory content results. Note that in the specified vertical memory location denoted by destination address a copy of the result will be deposited.

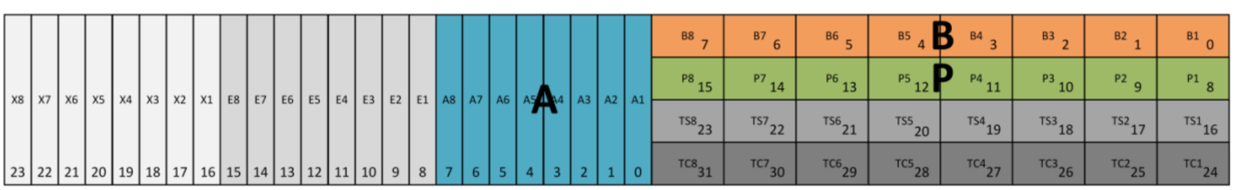
The outcome **C** can be read via device driver from memory, transferred to the application and shown on command line.



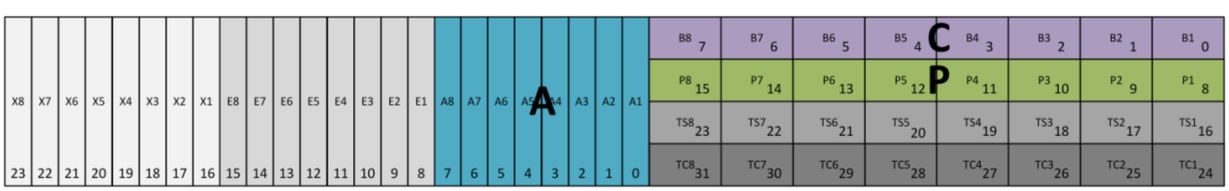
We save the result in X memory location, we need to compare it with R and N-R values.

```
// ===== if temporary == r or temporary == n-r: continue else return false ====
// temporary != r
u32 temporaryNotEqualR = 0;
for(i=0; i<word_count; i++) {
    if(temporary[word_count-1-i] != r[word_count-1-i]) {
        temporaryNotEqualR = 1;
        break;
    }
}
// temporary != n - r
u32 temporaryNotEqualNMinR = 0;
for(i=0; i<word_count; i++) {
    if(temporary[word_count-1-i] != NSubR[word_count-1-i]) {
        temporaryNotEqualNMinR = 1;
        break;
    }
}
// if temporary != r and temporary != n-r
if(temporaryNotEqualR == 1 && temporaryNotEqualNMinR == 1) {
    for(k=0; k < s; k++) {
    // ============= temporary = (temporary * temporary * rinv) % n =============

    // Write value of 'temporary' from temporary[128] to A register
        for(i=0; i<word_count; i++) {
            iowrite32(temporary[word_count-1-i], (MWMAC_RAM_ptr+0x200+i));
        }

    // Write value of 'temporary' from temporary[128] to B register
        for(i=0; i<word_count; i++) {
            iowrite32(temporary[word_count-1-i], (MWMAC_RAM_ptr+0x3+i*0x4));
        }

        // MontMult(A, B, P)
        //           Start      Abort        f_sel      sec_calc         precision          operation
        mwmac_cmd = (1 << 0) | (0 << 1) | (mwmac_f_sel << 2) | (mwmac_sec_calc << 3) | (mwmac_precision << 4) | (MONTMULT << 8)
                //           src_addr                  dest_addr            src_addr_e   src_addr_x
                | (MWMAC_RAM_B1 << 12) | (MWMAC_RAM_A1 << 17) | (0x0 << 22) | (0x0 << 27);
        iowrite32(mwmac_cmd, MWMAC_CMD_ptr);
        // Wait until CryptoCore is busy
        while (!mwmac_irq_var);
        mwmac_irq_var = 0;

        // Copying value result 'temporary' from B register to temporary[128];
        for (i = 0; i < word_count; i++) {
            temporary[word_count - 1 - i] = ioread32(MWMAC_RAM_ptr + 0x3 + i * 0x4);
        }
```

So, we start by implementing MontMult and save the result to the X. So, at the beginning we copy the X to the A and B and then we run the MontMult (A, B, P) syntaxes. Again, compare the result with **R** and **N-R** and continue based on the algorithm conditions.



The outcome **C** can be read via device driver from memory.



And as a final check we make sure that if the value is equal to R or the number of iterations in the loop is equal to the number of shifts then it is composite, otherwise it should complete all the test cases and return as probably prime number.

```
            // if temporary == r: return False
            u32 temporaryEqualR = 1;
                for(i=0; i<word_count; i++) {
                    if(temporary[word_count-1-i] != r[word_count-1-i]) {
                        temporaryEqualR = 0;
                        break;
                    }
                }
                if(temporaryEqualR == 1) {
                    MRT_prime_params_ptr->probably_prime = 0;
                    return;
                }
            // if temporary == n-r: break
            u32 temporaryEqualNMinR = 1;
            for(i=0; i<word_count; i++) {
                    if(temporary[word_count-1-i] != NSubR[word_count-1-i]) {
                        temporaryEqualNMinR = 0;
                        break;
                    }
                }
                if(temporaryEqualNMinR == 1) {
                    break;
                }
            }
            if (k == s) {
                MRT_prime_params_ptr->probably_prime = 0;
                return
            }

        }
    }
    MRT_prime_params_ptr->probably_prime = 1;
```