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.

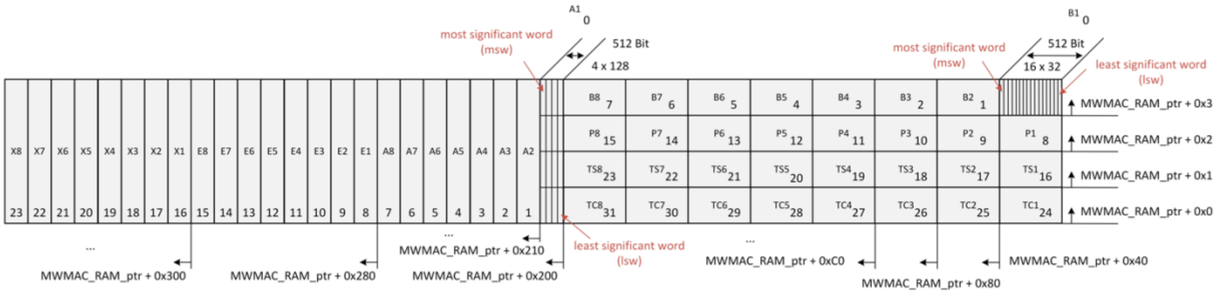


Figure 1: Montgomery Multiplication Core RAM Organization

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.

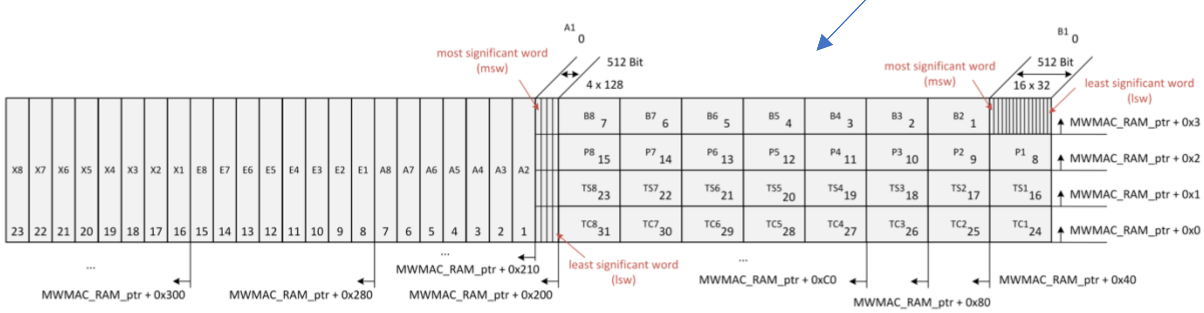
The locations named A, E and X for convenience usually are used to hold operand A in Montgomery Multiplication and Montgomery Exponentiation operations as well as the exponent operand E and the auxiliary operand X in Montgomery Exponentiation operations. In addition, all RAM slots are intended to hold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

Implementation on CryptoCore

```
def miller_rabin(n, k = 40, isMontgomery = False):

    s = 0
    d = n-1
    prec = n.nbits()
    r = (2^prec) % n
    rinv = inverse_mod(r,n)

    while d & 1 == 0:
        d = d >> 1
        s += 1

    for _ in range(k):
        a = random.randrange(2, n-2)
        if isMontgomery:
            temporary = Prime_MontExp(a,d,n,prec,r,rinv)
            if temporary == r or temporary == n-r:
                continue
            for _ in range(s - 1):
                temporary = (temporary * temporary) % n
                if temporary == n - 1:
                    break
            else:
                return False
        else:
            temporary = Prime_ModExp(a,d,n,prec)
            if temporary == 1 or temporary == n-1:
                continue
            for _ in range(s - 1):
                temporary = (temporary * temporary) % n
                if temporary == n - 1:
                    break
            else:
                return False
    return True
```

So, after understanding the RAM management in the current application we are going to talk about, how we are going to implement the current code (MRT in Montgomery domain) in CryptoCore.

Following figure given an overview of required offsets in order to access specific memory addresses, and exemplified highlights the definition of least significant word (lsw) and most significant word (msw).



Dive into the /driver file

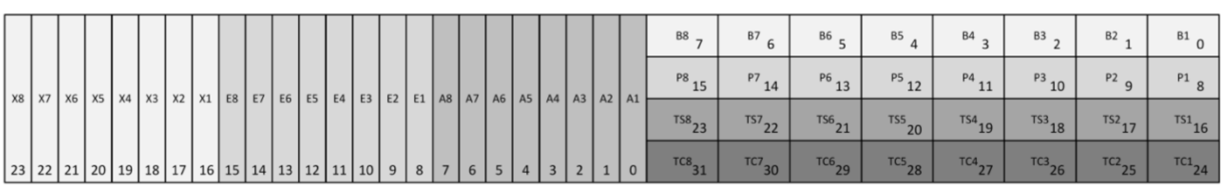
Part 1

```
def miller_rabin(n, k = 40, isMontgomery = False):

    s = 0
    d = n-1
    prec = n.nbits()
    r = (2^prec) % n
    rinv = inverse_mod(r,n)

    while d & 1 == 0:
        d = d >> 1
        s += 1

    for _ in range(k):
        a = random.randrange(2, n-2)
        if isMontgomery:
            temporary = Prime_MontExp(a,d,n,prec,r,rinv)
            if temporary == r or temporary == n-r:
                continue
            for _ in range(s - 1):
                temporary = (temporary * temporary) % n
                if temporary == n - 1:
                    break
            else:
                return False
        else:
            temporary = Prime_ModExp(a,d,n,prec)
            if temporary == 1 or temporary == n-1:
                continue
            for _ in range(s - 1):
                temporary = (temporary * temporary) % n
                if temporary == n - 1:
                    break
            else:
                return False
    return True
```

In this algorithm, we have one main for loop and before the for loop there are some mathematical calculations like shifting the input (n) and calculate (d) and (s).

In addition we calculate the (r) and (n-r) that we use them a lot in the algorithm.

Coming from an empty or rather unknown memory content.



Part 2

```
def miller_rabin(n, k = 40, isMontgomery = False):

    s = 0
    d = n-1
    prec = n.nbits()
    r = (2^prec) % n
    rinv = inverse_mod(r,n)

    while d & 1 == 0:
        d = d >> 1
        s += 1

    for _ in range(k):
        a = random.randrange(2, n-2)
        if isMontgomery:
            temporary = Prime_MontExp(a,d,n,prec,r,rinv)
            if temporary == r or temporary == n-r:
                continue
                for _ in range(s - 1):
                    temporary = (temporary * temporary) % n
                    if temporary == n - 1:
                        break
            else:
                return False
        else:
            temporary = Prime_ModExp(a,d,n,prec)
            if temporary == 1 or temporary == n-1:
                continue
                for _ in range(s - 1):
                    temporary = (temporary * temporary) % n
                    if temporary == n - 1:
                        break
            else:
                return False
    return True
```

The main loop will run (k) times.

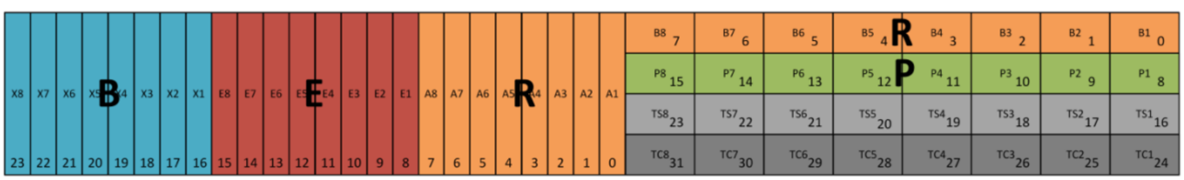
The first task in the main for loop is calculating Montgomery exponentiation.

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).

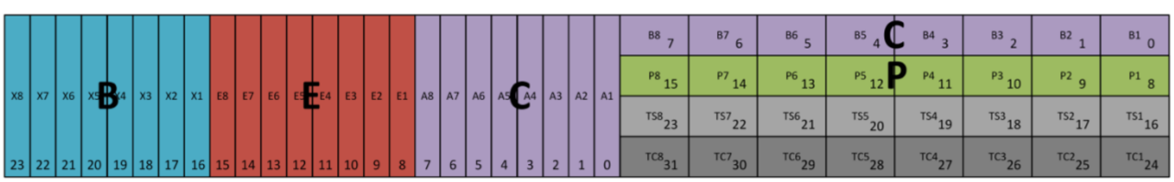
We save the result to the X and compare it with R and N-R by for loops in the code.

```
def Prime_MontExp(b,e,n,prec,r,rinv):
  r2 = (r*r) % n
  x = (1 * r2 * rinv) % n
  exp = e
  for i in reversed(xrange(prec)):
    x = (x * x * rinv) % n
    if(Integer(exp).digits(base=2,padto=prec)[i] == 1):
      x = (b * x * rinv) % n
  c = x
  return(c)
```

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



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



Part 3

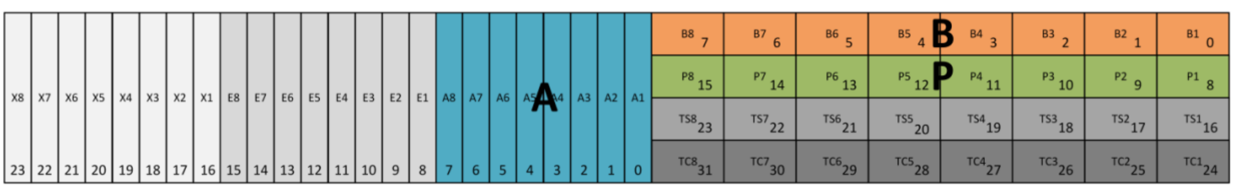
```python
def miller_rabin(n, k = 40, isMontgomery = False):

    s = 0
    d = n-1
    prec = n.nbits()
    r = (2^prec) % n
    rinv = inverse_mod(r,n)

    while d & 1 == 0:
        d = d >> 1
        s += 1

    for _ in range(k):
        a = random.randrange(2, n-2)
        if isMontgomery:
            temporary = Prime_MontExp(a,d,n,prec,r,rinv)
            if temporary == r or temporary == n-r:
                continue
            for _ in range(s - 1):
                temporary = (temporary * temporary) % n
                if temporary == n - 1:
                    break
            else:
                return False
        else:
            temporary = Prime_ModExp(a,d,n,prec)
            if temporary == 1 or temporary == n-1:
                continue
            for _ in range(s - 1):
                temporary = (temporary * temporary) % n
                if temporary == n - 1:
                    break
            else:
                return False
    return True
```

In this step we should implement MontMult and save the result to the temporary variable.

So at the beginning we copy the temporary 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

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



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