Technical notes on using Analog Devices products and development tools
Visit our Web resources http://www.analog.com/ee-notes and http://www.analog.com/processors or e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support.

Implementing the RSA Cryptosystem with the Public Key Accelerator

Contributed by G. Yi *Rev 1 – October 14, 2015*

Introduction

Along with the full rich set of system peripherals, the ADSP-BF70x Blackfin+™, ADSP-SC58x SHARC+™ and ADSP-2158x SHARC+ processors include cryptographic hardware engines. This EE-Note demonstrates the use of some of these engines through the implementation of the RSA cryptosystem.

This document describes:

- Overview of the RSA cryptosystem
- Setting up and using the cryptographic hardware accelerators
- Implementation of the RSA cryptosystem
- Performance of the cryptographic hardware accelerators

RSA Cryptosystem

RSA is a public-key cryptosystem defined in the *PKCS #1 v2.2: RSA Cryptography Standard*[1]. A public-key crypto system consists of two keys, forming a key pair. When a user generates a key pair, one of the keys is held private. The other key is public, which can be freely distributed.

Protection

The RSA cryptosystem can be used for both confidentiality protection, where information is protected from unauthorized users, as well as guaranteeing authenticity and integrity of the information.

Confidentiality

To use RSA for confidentiality protection, the distributed public key is used to encrypt a message. The encrypted message can then be transmitted via a non-secure communication channel, and only the user(s) holding the private key will be able to decrypt the message and obtain the original content.

Authenticity and Integrity

RSA can also be used to help authenticate and ensure the integrity of a message. In this case, digital signatures are employed. To generate a signature, the private key is to "sign" a message (or hash digest of the message). The digital signature is then attached to the original message and sent to the destination. To prove the authenticity and integrity of the message, the public key is then used to verify the signature. If the signature can't be validated, the user will know that either the signature and/or the message was corrupted or the public key used for the signature verification was also incorrect.

Procedure

The following sections describe the methodology for using the RSA cryptosystem.

Generating a Prime Modulus

The first step of the RSA is to create the key pair. In order to do this, a prime modulus (n) must be generated It's composed of two or more prime factors.

$$n = p0 \cdot p1 \cdot ...$$

Equation 1. Calculating Prime Modulus

Equation 1 shows how to calculate n. As seen, it's a multiplicative product of distinct odd primes, $p0$, $p1$, etc.

In the code contained in the *associated ZIP file*[2], n is calculated using only two primes, p and q. The security of RSA relies on the ability to factor n into its prime factors. Therefore, the larger the value of n, the harder it is to perform the factorization. A popular size of n for today's security needs is 1024 bits. The size/bit length is ultimately the user's decision based on the security requirements and threat models. For this example code, 64 bits was chosen. This can be changed via the `MOD_SIZE_BITS` macro defined in the `RSAexample.h` header file.

Generating Prime Numbers

In order to generate a large prime number, the example code uses the True Random Number Generator (TRNG) to generate a random number of a specified bit length, and then that number is tested to see if it's prime. If it's not prime, another number is generated.

Using the True Random Number Generator (TRNG)

The TRNG is comprised of eight free running oscillators (FRO) running at different frequencies. Bits are sampled and stored in an internal buffer. Once the buffer is full and bits are ready, they can be read out via output registers.

The TRNG also contains in-circuit testing for different patterns and runs. It will automatically start shutting down FROs if the in-circuit testing results in patterns or runs, making the output non-random. Thresholds can be configured to trigger alarms and generate an interrupt if too many FROs have been shut down. This would allow the user to take appropriate action, such as disregarding collected bits that aren't really random and detuning/restarting an FRO. Further details can be obtained in the *Processor Hardware Reference Manuals*[3][4].

In the example code, the TRNG is enabled, and the random numbers available in the output registers are read out. Once all the registers are read and more bits are needed, an acknowledgement bit is set to tell the TRNG to provide more random numbers in the output register. This is done in a loop until all the bits needed are obtained. The example only needs random numbers with relatively small bit sizes, so no elaborate scheme was implemented for error handling in case FROs started to shut down due to pattern-detection.

Primality Testing

There are different methods for primality testing, but a simple one was employed in the example code. Testing if the number is divisible by 2 is simply testing the least significant bit. To test if the number is divisible by 3, a state machine is employed. Transitions are made after analyzing every bit from the LSB to

the MSB. After 3, an iterator, i, starts at 5 and increments by 6 while the square of it remains less than the random number. For every iteration, it's tested to see if i or $i + 2$ divides the random number. If either does divide the random number, then it is a factor and the random number is not prime. Once all the iterations have completed - and none of the candidates divides the random number - then the random number can be deemed prime.

As one can imagine, for larger modulus sizes, more iterations need to be done, which can be quite time consuming. There are other optimized and probabilistic methods for primality testing and for prime number generation, but that is outside the scope of this EE-Note. Also, depending on the application requirements for the frequency of generating keys, a list of pre-generated large primes can be stored securely on the processor, at which point choosing a large prime would just be a matter of selection.

Using the Public Key Accelerator (PKA)

The example code starts to use the PKA during the primality testing. The following math functions are used:

- Modulo – to test if i or $i + 2$ can divide the test number
- Comparison – to test if i^2 is less than the test number
- Multiplication – to calculate i^2
- Addition – to calculate $i + 2$
- Subtraction – to help with manual division

Overview of PKA

The PKA is a big number arithmetic engine. Since the security of public key algorithms relies on the difficulty of factoring large numbers, the use of these algorithms employ bit lengths ranging from hundreds to thousands of bits. For RSA using 1024-bit modulus, a number would use 32 words (of width 32-bit).

Processing such large numbers would be time consuming if done by software running on the Blackfin+ core. The PKA helps offload this computational workload, as it offers support for basic mathematical functions such as multiply, divide, add, subtract, shift, compare, copy, and modulo. It also offers support for more complex functions like modular exponentiation and inversion. Finally, the PKA also offers support for functions pertaining specifically to elliptic curve cryptography, which includes ECC addition and ECC multiply.

Using the PKA is fairly straightforward. It contains a 4KB RAM, whose start address coincides with the `PKA0_RAM` memory-mapped register (MMR). The input values are loaded into this memory, and the pointers are configured to indicate to the PKA where the values are in the RAM. The size MMRs are also configured to tell the PKA how big the inputs are. The output pointer MMR is also configured to tell the PKA engine where to place the result in RAM. `PKA0_APTR` and `PKA0_BPTR` always point to inputs, while `PKA0_CPTR` and `PKA0_DPTR` will point to inputs or results, depending on what operation the PKA is being configured to execute, as shown in Figure 1.

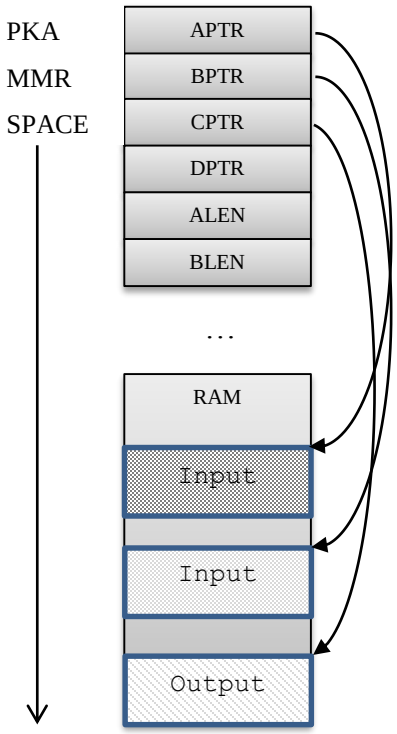

Figure 1- PKA Memory Space Usage

Next, the math function is configured, and the execution is started. A status bit is then polled for completion.

Listing 1 is a code snippet showing how to set up the PKA to perform multiplication. The input parameters to the function are `vectA` and `vectB`, both pointers to buffers of unsigned 32-bit integers, which hold the large numbers. The lowest addresses, `vectA[0]` and `vectB[0]`, hold the least significant words of the large numbers, while the highest addresses hold the most significant words. Additionally, the inputs `vectAsz` and `vectBsz` indicate the size of the buffers in terms of unsigned 32-bit words.

These inputs lengths can be written directly to the `PKA0_ALEN` and `PKA0_BLEN` MMRs because the PKA expects the lengths to be configured in terms of 32-bit words.

Next, the PKA RAM is partitioned to set up where the input will reside and where the PKA will place the resulting output. This is accomplished by writing to the `PKA0_APTR`, `PKA0_BPTR`, and `PKA0_CPTR` pointer MMRs, which are basically indices into the 32-bit buffers in the PKA RAM. For multiplication, the PKA requires that the input and output buffers need to be 8-byte aligned. Therefore, the code sets the `PKA_BPTR` and `PKA_CPTR` registers by first adding 1 to `vectAsz` to round up, and then clearing the least significant bit.

Once the lengths and pointers are set up, the inputs are copied from the buffers to the PKA RAM in their allocated regions. Then, the `PKA0_FUNC` MMR is configured for the operation and set to let the PKA execute.

The '`RUN`' bit in the `PKA0_FUNC` can be polled to determine if the PKA is finished processing. If it is, the result can then be copied out of the PKA RAM. For a multiplication, the maximum output bit size is the sum of the size of the multiplicand and the size of the multiplier. The result size may be less, depending on the values of the multiplicand and the multiplier. For this reason, the `PKA_MSW` MMR will indicate where

the most significant word that is non-zero is in PKA_RAM. This can also be used for the other basic arithmetic operations that the PKA supports.

```
uint32_t *pka_mult(uint32_t *vectA, uint32_t vectAsz, uint32_t *vectB, uint32_t
                      *vectBsz, uint32_t *result){
    .
    /* configure lengths of input */
    *pREG_PKA0_ALEN = vectAsz ;
    *pREG_PKA0_BLEN = vectBsz ;

    /* configure pointers to where input and output will be in PKA RAM */
    *pREG_PKA0_APTR = 0 ;
    *pREG_PKA0_BPTR = *pREG_PKA0_APTR + ((vectAsz+1)&(0xfffffffe)) ;
    *pREG_PKA0_CPTR = *pREG_PKA0_BPTR + ((vectBsz+1)&(0xfffffffe)) ;

    /* copy input vectors into PKA RAM */
    for(i=0; i<vectAsz; i++)
         pREG_PKA0_RAM[*pREG_PKA0_APTR + i] = vectA[i];

    for(i=0; i<vectBsz; i++)
         pREG_PKA0_RAM[*pREG_PKA0_BPTR + i] = vectB[i];

    /* configure operation and start */
    *pREG_PKA0_FUNC = BITM_PKA_FUNC_MULT | BITM_PKA_FUNC_RUN;

    /* poll 'run' bit in function register and wait for completion */
    .
    .
    <wait for operation to complete>
    .
    .

    /* copy result from PKA ram to output buffer */
    for(i=0; i<(vectAsz+vectBsz); i++)
         result[i] = pREG_PKA0_RAM[*pREG_PKA0_CPTR + i];

    return result;
}
```

Listing 1 - Example Code to Setup PKA for Multiplication

For the associated example RSA code, a simple API was created to use the PKA for the various large number math functions, which can be found in the pka_functions.c source file. The functions are blocking, meaning they wait until the operation is completed and no interrupts are used. One can envision a more involved and complex usage of the PKA, depending on the application, involving pre-loading multiple inputs, using interrupts, overlapping inputs and outputs, automatically setting the pointers to use output as input, etc.

For more detailed information on the PKA, refer to the hardware reference manual.

Generating the Encryption Key

At this point, the TRNG and PKA have been used to two generate prime numbers, which are multiplied to produce the prime modulus.

Next, Euler's phi function is calculated for the prime modulus n. For an input number, Euler's phi function provides the count of the number of values which are less than or equal to the input and are relatively prime to the input value. Since Euler's phi function is multiplicative, it can be found by $(n) = \varphi(p_0)\varphi(p_1) \ldots$, etc. Since prime numbers are the function's inputs, the results are $\varphi(p_0) = p_0 - 1, \varphi(p_1) = p_1 - 1$, etc. Therefore, $\varphi(n) = (p_0 - 1)(p_1 - 1) = (p - 1)(q - 1)$, since the example code uses only two primes to generate the prime modulus.

Again, since all the numbers being used are large numbers, the PKA is used to perform the subtractions and multiplication.

For a sanity check, Euler's phi function can alternatively be calculated with:

$$(p - 1)(q - 1) = pq - p - q + 1 = n - (p + q - 1)$$

Once Euler's phi function is calculated, another random number, e (the public key exponent), is generated satisfying the following properties:

1. $1 < e < \varphi(n)$, and
2. $GCD(e, \varphi(n)) = 1$

The PKA is used to perform the comparison in #1 above and the greatest common divisor (GCD) test in #2 above. The modular inversion function will return an error indication for two reasons:

- If the modulus is an even number.
- If the GCD of the number to invert and the modulus is not equal to one.

Therefore, the modular inversion function is set up with:

$\varphi(n) = number\ to\ invert$

and

$e = modulus$

This setup is important because the modulus cannot be an even number; and since, in this case, Euler's phi function is a product of even numbers, it will also be an even number.

Numbers are generated in a loop until one is found that satisfies the conditions listed above. Once one is found, this encryption key (also known as the public key) is the pair consisting of the public key exponent and the prime modulus, (e, n).

Generating the Decryption Key

Once the public key exponent is found, the decryption key, d, is calculated by taking the inverse of e.

$d \equiv e^{-1} mod\ (\varphi(n))$ or $de \equiv 1\ mod(\varphi(n))$

The problem here is that $\varphi(n)$is an even number because p and q are odd prime numbers. When the modular inversion function in the PKA is set with an even number for the modulus, an error results.

To solve this, an alternative calculation is used:

$$d \equiv ModInv\ (e, \varphi(n)) \equiv (1 + (\ \varphi(n) * \big(e - ModInv(\ \varphi(n), e)\big))/e$$

So, with the PKA, in order to find the decryption key, the modular inversion operation is still used, but the operands are reversed and a few more operations are required.

When the modular inverse of e is found, the decryption key is the pair (d, n).

The alternate form of the decryption key is a 5-tuple of $(p, q, dP, dQ, qInv)$ where:

$dP\ =\ d\ mod\ (p-1)$,

$dQ\ =\ d\ mod\ (q-1)$, and

$qInv\ =\ q^{-1}\ mod\ p$.

With this form, the Chinese Remainder Theorem (CRT) is used to optimize the calculation. Details and proof of the CRT is outside the scope of this EE-Note.

Encryption/Decryption

Now that the keys have been generated, the public key is distributed and is available for anyone to use.

To encrypt a message, it's just a matter of modular exponentiation. Let m be the message, and let c be the encrypted message $m^e mod\ n$.

Again, the PKA can be used to calculate this. The constraints on m are:

1. $0 \leq m < n$
2. GCD(m,n)=1

For the example code, m is just a generated random number, and again the modular inversion function from the PKA is used to test if the GCD between the message and the prime modulus is 1.

The encrypted message, or ciphertext, can then be transmitted to the receiver, who holds the private key, (d, n). Decrypting the message is also a matter of modular exponentiation:

$$m = c^d mod\ n$$

Therefore, if the prime modulus and keys were generated correctly, the original message will be the result.

If the alternate form of the private key (the 5-tuple) is used, the original message would be obtained by calculating:

$m_1 = c^{dP}$,

$m_2 = c^{dQ}$,

$h = (m_1 - m_2) \cdot qInv\ mod\ p$,

and then finally $m =\ m_2 + q \cdot h$.

This might seem to be a lot of calculations and preparation for the PKA engine; but, as the example code also demonstrates, the PKA engine natively supports the use of this alternative form of the private key. This is done by loading the elements from the 5-tuple and the ciphertext into the appropriate locations, and then trigger it to start calculating.

Signature Generation and Verification

For confidentiality, the public key is used to encrypt, and then the private key is used to decrypt. For integrity and authenticity, the keys are used in reverse. Also, as is the case with encryption and decryption, it's a matter of modular exponentiation. The sender uses the private key to encrypt what is typically a hash digest of the message to create a digital signature.

$$signature = (hash(message))^d mod\ n$$

This signature is sent along with the message to the receiver. The receiver, who has the public key, then uses it to verify the signature by raising the signature to the power of the private key exponent, mod n.

$$signature^e\ mod\ n =? hash(message)$$

The receiver also computes the hash digest of the message. This hash digest is then compared against the result of the modular exponentiation. If the two values match, then the signature is verified. If they don't match, then either the signature or message was changed during transmission, or the public key is invalid.

RSA Example Project

Throughout this EE-Note, the example project in the associated ZIP file has been referenced. The source code can be found in the `RSAexample\src` directory, comprised of the following files:

1. `RSAexample.c/h` – contains the main application and functions directly associated with the RSA cryptosystem (i.e., `genKeyPair()`, `genPrimNum()`, `rsaEncrypt()`, `rsaDecrypt()`, etc.
2. `math_utilities.c/h` – contains math helper functions such as `aDivB()` to handle large number division when the divisor is a 32-bit number. This is used because the PKA only performs large number division when the divisor is greater than 32-bits. It also contains the functions to test if a number is prime, `isPrime()`, and also a function to calculate the modular inverse of a number, `getModInv()`.
3. `pka_functions.c/h` – contains a simple driver API to use various simple PKA functionality.

In `RSAexample.h`, the `DEBUG` macro is defined. It can be set with a value up to 3 to output various levels of debug information.

The code also builds and runs on the ADSP-BF706, and the ADSP-SC589, separately on the ARM and the SHARC cores. The project files are found in the `RSAexample/projects` directory.

PKA Benchmarks

Benchmarks are provided in Table 1 (for ADSP-BF70x Blackfin+ processors) and Table 2 (for ADSP-2158x/ADSP-2158x SHARC+ processors) for RSA operations, mainly modular exponentiations. For each operation, there are two benchmarks. One is using one odd power, and the other is using four odd powers. The number of odd powers to pre-calculate and use during the modular exponentiation is configurable by writing to the `PKA_SHIFT` register. The decision on which to use is a trade-off between speed and memory (the PKA RAM). The more odd powers that are pre-computed and used, the faster the modular exponentiation will be because less multiplications are needed. There is a limit though, and a higher number of odd powers would not be advisable for smaller bit lengths because the process of pre-computing all the

odd powers also requires a certain amount of computation time. The balance between the number of odd powers used and the key bit length will have to be decided by user.

Operation	HW operation	Exponent Size (bits)	Modulus Size (bits)	# of Odd Powers	Performance (cycles)
RSA sign/encr/decr 512	Modular exponentiation	512	512	1	690,551
RSA sign/encr/decr 512	Modular exponentiation	512	512	4	588,939
RSA sign/encr/decr 1024	Modular exponentiation	1024	1024	1	4,099,965
RSA sign/encr/decr 1024	Modular exponentiation	1024	1024	4	3,468,801
RSA sign/encr/decr 2048	Modular exponentiation	2048	2048	1	29,420,846
RSA sign/encr/decr 2048	Modular exponentiation	2048	2048	4	24,561,385
RSA sign/encr/decr 4096	Modular exponentiation	4096	4096	1	214,598,077
RSA sign/encr/decr 4096	Modular exponentiation	4096	4096	4	180,135,856
RSA sign/decr CRT 512	2x Modular exponentiation	256	256	1	240,733
RSA sign/decr CRT 512	2x Modular exponentiation	256	256	4	208,759
RSA sign/decr CRT 1024	2x Modular exponentiation	512	512	1	1,397,951
RSA sign/decr CRT 1024	2x Modular exponentiation	512	512	4	1,170,332
RSA sign/decr CRT 2048	2x Modular exponentiation	1024	1024	1	8,344,143
RSA sign/decr CRT 2048	2x Modular exponentiation	1024	1024	4	6,941,292
RSA sign/decr CRT 4096	2x Modular exponentiation	2048	2048	1	59,514,54
RSA sign/decr CRT 4096	2x Modular exponentiation	2048	2048	4	49,334,463

Table 1. PKA Benchmarks for RSA Operations for ADSP-BF70x Blackfin+ Processors

Also note that when the Chinese Remainder theorem is used, it saves the computation cycles by almost a factor of three for the equivalent bit lengths. That is because doing two modular exponentiations on half the bit length is computationally less expensive than doing one modular exponentiation on the original bit length size.

Finally, the difference in the performance between the PKA on the ADSP-BF70x Blackfin+ processor and the ADSP-SC58x/ADSP-2158x SHARC+ processor is due to the width of the processing elements. On the ADSP-BF70x processor, the PKA is 32-bit, while the ADSP-SC58x/ADSP-2158x processor PKA is 16-bit.

Operation	HW operation	Exponent Size (bits)	Modulus Size (bits)	# of Odd Powers	Performance (cycles)
RSA sign/encr/decr 512	Modular exponentiation	512	512	1	2,299,238
RSA sign/encr/decr 512	Modular exponentiation	512	512	4	1,959,458
RSA sign/encr/decr 1024	Modular exponentiation	1024	1024	1	14,779,279
RSA sign/encr/decr 1024	Modular exponentiation	1024	1024	4	12,501,019
RSA sign/encr/decr 2048	Modular exponentiation	2048	2048	1	111,325,297
RSA sign/encr/decr 2048	Modular exponentiation	2048	2048	4	92,931,362
RSA sign/encr/decr 4096	Modular exponentiation	4096	4096	1	833,993,111
RSA sign/encr/decr 4096	Modular exponentiation	4096	4096	4	700,049,874
RSA sign/decr CRT 512	2x Modular exponentiation	256	256	1	704,406
RSA sign/decr CRT 512	2x Modular exponentiation	256	256	4	608,979
RSA sign/decr CRT 1024	2x Modular exponentiation	512	512	1	4,643,494
RSA sign/decr CRT 1024	2x Modular exponentiation	512	512	4	3,882,581
RSA sign/decr CRT 2048	2x Modular exponentiation	1024	1024	1	30,055,168
RSA sign/decr CRT 2048	2x Modular exponentiation	1024	1024	4	24,992,042
RSA sign/decr CRT 4096	2x Modular exponentiation	2048	2048	1	225,147,137
RSA sign/decr CRT 4096	2x Modular exponentiation	2048	2048	4	186,614,169

Table 2. PKA Benchmarks for RSA Operations on ADSP-SC58x/ADSP-2158x SHARC+ Processors

Disclaimer

The intended purpose of this EE-Note and associated code was to demonstrate the use of the hardware accelerator engines in the implementation of the RSA cryptosystem. The code is provided as reference only. Analog Devices, Inc. does not make any claims that this code is bug-free nor suitable for secure applications. It is left to the user's discretion on how the code is incorporated into the end product/application.

References

[1] *PKCS #1 v2.2: RSA Cryptography Standard*. v2.2, October 27, 2012. RSA Laboratories.

[2] *Associated ZIP File for EE-385*. Rev 1, October, 2015. Analog Devices, Inc.

[3] *ADSP-BF70x Blackfin+ Processor Hardware Reference*, Preliminary Revision 0.2, May 2014, Analog Devices, Inc.

[4] *ADSP-SC58x SHARC Processor Hardware Reference*, Preliminary Revision 0.2, June 2015, Analog Devices, Inc.

Document History

Revision	Description
Rev 1 – October 14, 2015 *by G.Yi*	Initial Release