**Intermediate Reports on DSP Optimizations**

**Group Name** : crypto group 1

**Team member 1** : Axel Vanraes

**Team member 2** : Michaël Raes

Michaël Raes Axel Vanraes

 

Include the following details in your reports.

 Optimizations tried and carried out. Discuss its impact on the cycles at the function level and total code level. **Also present your reasoning** about the impact of the optimization which you observed.

 Even if an optimization does not improve the result, include and discuss it.

The following table is an overview of all performed optimizations. (**KA**=Key Agreement, **ED**=Encrypt/Decrypt)  
The input data (a 30 byte buffer) is defined in a variable (during phase 1)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Major Optimization | Functions Effected | Total Cycles\* |  | Change (in %) |
|  |  |  | **Before(KA/ED)** | **After** |  |
| Base Code | N/A | N/A | 537M / 49K | N/A | N/A |
| Session 1 | S1.keygetters | get\_slave\_privkey  get\_slave\_pubkey  get\_slave\_pubkey  get\_master\_privkey  get\_master\_pubkey  get\_master\_modulus | 537M / 49K | 537M / 49K | 0.00025 |
|  | S1.modinv | mod\_inverse | 503K | 400 | 99.92% |
| Session 2 | S2.modinv | Mod\_inverse | 503K | 1190 |  |
|  | S2.Must\_It | Monmult | 535M | 535M | 0 |
|  | S2.unroll | Monmult | 535M | 513M | 4.11% |
| Session 3 | S3.0xff | Monmult | 513 M | 513M | 0% |
| : | : | : |  |  |  |

# Report 1

**Lab date : 22/04, 2016  
  
Submission date : 23/04, 2016**

**Submission time : 4 PM**

## Logs:

This session was used for profiling the base code and identifying the most obvious optimizations. It is clear from the profiling results that the only significant gain is to be expected from optimizations to the Montgomery multiplication code. Minor improvements can be expected from internally restructuring code to avoid the needless endianness conversions we do.

Some final modifications were made on the functionality (EOT, reset of symmetric key, …) this may add some cycles to the asymmetric part but the effect is negligible.

## Profile results: (Subdirectory names are abbreviated. as=asymmetric, ci=ciphers, pm=packman, …) (monmult = Montgomery Multiplication, monexp = Montgomery Exponentiation)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Function Name | File Name | Total no. of calls | Exclusive Count Total\* | After  (This Session) | %Change | Inclusive Count Total\* | After  (This Session) | % Change | Optimization carried out |
| monmult | as/monmult.c | 4057 | 534M |  |  | 534.6M |  |  |  |
| monexp | as/monexp.c | 9 | 104K |  |  | 534.7M |  |  |  |
| mod\_inverse | as/monmult.c | 4057 | 503K | 400 | 99.92% | 503K | 400 | 99.92% | S1.modinv |
| *Multiple* | sts/\*.c |  |  |  |  |  |  |  | S1.keygetters |

**Optimization details:** (Optimization names follow the format “S(session number).keyword”.)  
**S1.modinv** is the removing of the unnecessary calculations of the least significant byte of the inverse of the modulus. This was performed for every Montgomery Multiplication, whilst it only has to be calculated once for every used modulus. There are 3 moduli in this application (master, slave and Diffie-Hellman) so this gives an improvement of (4057-3)/4057 = 99.9% or about 500K cycles, which is still incomparable to the heavyweight that is Montgomery Multiplication.  
  
In **S1.keygetters** we define the keys for master and slave directly in the right format (Little Endian), instead of using the given (Big Endian) vectors from Toledo. This allows us to remove the “getter”-functions that took some unnecessary cycles. We expected an insignificant improvement from this, but the situation has actually worsened: it now takes more cycles. We don’t really understand what happened. If we add getters that makes a saving the “getters” again the total cycles increases again with 168 cycles/getter. Master and slave use four of 8\*168=1344 cycles or 0.00025%.

## Discussion (important)

 We expect the change from 8 to 32 bit in the Montgomery Multiplication algorithm to give a huge performance increase. We’ll try to implement this by next session. Subsequently it can be interesting to rewrite parts of ‘monmult’ in assembly as the function takes roughly 132k cycles per call.

* The performance of the real-time component (the encryption/decryption using AES-OCB) is sufficient, so we will focus on the asymmetric crypto calculations.

# Report 2

**Lab date : 28/04, 2016  
  
Submission date : 29/04, 2016**

**Submission time : 4 PM**

## Logs:

We had to do some extra changes to preserve the same functionality as before optimization S1.modinv. The result is that the total cycle count of modular inversion is now about 1k instead of the 400 we previously stated. However, the amount of calls is still reduced from 4057 to 14.

The work on changing Montgomery Multiplication from 8-bit word size to 16-bit continues, but it takes more time than previously estimated. We try to finish it by next session.

We have also started looking at the list of “mandatory/recommended” optimizations to see if they apply in our case, specifically the use of the different pragma’s. We tried using the MUST\_ITERATE pragma but the result was quasi nonexistent. UNROLL gave better results.

## Profile results: (Subdirectory names are abbreviated. as=asymmetric, ci=ciphers, pm=packman, …) (monmult = Montgomery Multiplication, monexp = Montgomery Exponentiation)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Function Name | File Name | Total no. of calls | Exclusive Count Total\* | After  (This Session) | %Change | Inclusive Count Total\* | After  (This Session | % Change | Optimization carried out |
| mod\_inverse | as/monmult.c | 14 | 503K | 1190 | - | 503K | 1190 |  | S2.modinv |
| monmult | as/monmult.c | 4057 |  |  | 0 |  |  | 0 | S2.Must\_It |
| monmult | as/monmult.c | 4057 | 534M | 510M | 5% | 534M | 510M | 5% | S2.unroll |

**Optimization details:** (Optimization names follow the format “S(session number).keyword”.)  
**S2.modinv** is an update to the changes we did in S1.modinv to iron out some bugs.  
In **S2.Must\_It** we tried to use the MUST\_ITERATE pragma for some of our Montgomery Multiplication loops, this gave no significant result. We’re still trying to figure out how to use pragma’s effectively.  
**S2.unroll**, where we did some loop unrolling in the Montgomery Multiplication, this gave a reduction of about 24M cycles, or about 5%.

## Discussion (important)

* We will change the Montgomery word size from 8 to 16, instead of the 32 previously stated, because multiplications with 16 bit words are the most efficient on the target DSP.
* An analysis of our Montgomery Multiplication (FIPS) algorithm requires 2s 2 + s multiplications, 6s 2 + 2s + 2 additions, 9s 2 + 8s + 2 reads, and 5s 2 + 8s + 1 writes, where **s** is the amount of “Montgomery words” in the modulus. For the non-optimized code **s** is 156, however if we change the word size to 16 bit, **s** is only 78. The vast possibility for cycle reduction is apparent.

# Report 3

**Lab date : 29/04, 2016  
  
Submission date : 30/04, 2016**

**Submission time : 4 PM**

## Logs:

We have now introduced a lot of flexibility in the Montgomery Multiplication code. It’s now possible to do the multiple-precision operations using words with an arbitrary (multiple of 1 byte) word length, so it could be ported to other architectures without having to do a huge amount of work. There is however still something wrong in the interfacing between exponentiation and multiplication. If the deadline for this report had been an hour or two later this would have been resolved. We will report on the performance of this crucial optimization in our next report.

The optimization documents for the C6000 core recommend that the programmer experiment with different factor value when using the UNROLL pragma, and pick the factor that yield the best performance. Loop unrolling can speed up but the consequence is that the size of the loop increases and it hurts the compiler. After a try and error process, the initial factor gives coincidentally the best results. So there no improvement with the last session on this topic.

## Profile results: (Subdirectory names are abbreviated. as=asymmetric, ci=ciphers, pm=packman, …) (monmult = Montgomery Multiplication, monexp = Montgomery Exponentiation)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Function Name | File Name | Total no. of calls | Exclusive Count Total\* | After  (This Session) | %Change | Inclusive Count Total\* | After  (This Session | % Change | Optimization carried out |
| monmult | as/monmult.c | 4057 | 510M | 510M | Approx. 0 | 510M | 510M | Approx. 0 | S3.0xff |

**Optimization details:** (Optimization names follow the format “S(session number).keyword”.)  
**S3.0xff** is the removal of unnecessary bit masking statements in the Montgomery code. It did not have a significant impact.

## Discussion (important)

* To minimize the overall iteration interval we have to minimize the loop carried dependency bound. One of the techniques suggested by TI is to use the restrict keyword. This might be very effective in the loops of ‘monmult.c’, we investigate this further.
* Furthermore, we studied the pipeline possibilities of the C6000 core. There are some possibilities to pipeline the kernel of the loops inside ‘monmult.c’ but this will take some time to think about and implement. First of all we have to learn the assembly instructions supported by C6x.
* If the DSP has a “multiply-accumulate” architecture we can exploit this in our FIPS (Finely Integrated Product Scanning) Montgomery Multiplication algorithm. Actually, the algorithm is most efficient of all comparable algorithms when you can store the core algorithm values t[1,2,3] in a single hardware accumulator and add the product from the FIPS loop directly to that accumulator. Again, more research into the ASM instructions of the processor is needed.