Implementation of an Elliptic Curve Cryptosystem on an 8-bit Microcontroller

Chris K Cockrum
Email: ckc@cockrum.net

Spring 2009

Abstract

This paper presents a study of the feasibility of an elliptic curve cryptosystem an 8-bit microcontroller as well as an example implementation. The cryptosystem is implemented using FIPS PUB 186-2 [3] as an exemplar. The focus of this paper is implementation efficiency.

Keywords: Microcontroller Implementation Elliptic Curve Cryptography Generalized Mersenne Prime

1 Introduction

An implementation of an elliptic curve cryptosystem on a Microchip PIC18F2550 microcontroller is outlined. The 8-bit bus width along with the data memory and processor speed limitations present additional challenges versus implementation on a general purpose computer. All algorithms required to perform an elliptic curve Diffie-Hellman key have been implemented.

2 System Description

This system will demonstrate the creation of a shared secret between the host PC and the embedded target (microcontroller).

2.1 Elliptic Curve

The chosen NIST curve is P-256 which uses the following elliptic curve over a prime field with prime p

$$y^2 = x^3 - 3x + 41058363725152142129326129780047268409114441015993725554835256314039467401291 \quad (1)$$

$$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 = 115792089210356248762697446949407573530086143415290314195533631308867097853951 \quad (2)$$

Which has order:

$$r = 115792089210356248762697446949407573529996955224135760342422259061068512044369 \quad (3)$$

For this paper, the following base point will be used:

$$G = (48439561293906451759052585252797914202762949526041747995844080717082404635286, 36134250956749795798585127919587881956611106672985015071877198253568414405109) \quad (4)$$

2.2 Elliptic Curve Diffie-Hellman

The Diffie-Hellman key exchange creates a shared secret between two communicating parties. Both parties have agreed on the choice of elliptic curve, underlying field, and base point and these are made public. To create a shared secret, both parties (Alice and Bob) generate a random value in the range $(1, r-1)$ where r is the order of the elliptic curve.

$S_a =$ Alice's Secret Key
$S_b =$ Bob's Secret Key

Alice and Bob then calculate their public key by multiplying their respective secret numbers times the chosen base point on the curve.

$P_a = S_a(G_x, G_y) =$Alice's Public Key
$P_b = S_b(G_x, G_y) =$Bob's Public Key

Alice and Bob then exchange public keys. Now Alice and Bob multiply each other's public key with their secret key.

$Shared_a = S_a S_b(G_x, G_y) =$Shared secret as calculated by Alice
$Shared_b = S_b S_a(G_x, G_y) =$Shared secret as calculated by Bob

Since this operation is commutative, the shared secrets calculated by Alice and Bob are the same.

$Shared_a = Shared_b$

During this exchange, only the public keys are visible by anyone other than Alice and Bob. So the adversary needs to calculate the discrete logarithm of an element. The elliptic curve discrete logarithm problem is as follows.

Given two points, P and B on an elliptic curve, find an integer s such that $P = sB$

At the current time, this problem is believed to be intractable given a properly constructed elliptic curve with a sufficiently large order.

2.3 Hardware Design

The hardware was designed to be a small printed circuit board with minimized component count that operates from a personal computer's universal serial bus (USB) port. The hardware communicates and draws power solely from this connection.

2.3.1 Digital Circuit Design

To minimize the processing time, the hardware circuit was designed with a clock rate of 48 Megahertz (MHz) which is the maximum clock rate of the PIC18F2550 microcontroller. This microcontroller has an internal USB interface with minimal external parts required which led to a simplified design as shown in Figure 1

2.3.2 Random Number Generator

As the PIC18F2550 microcontroller and most other microcontrollers do not contain a random number generator, it is necessary to either obtain random numbers from another source or incorporate a hardware random number generator into the design. For cryptographic uses, it is very important that random numbers are

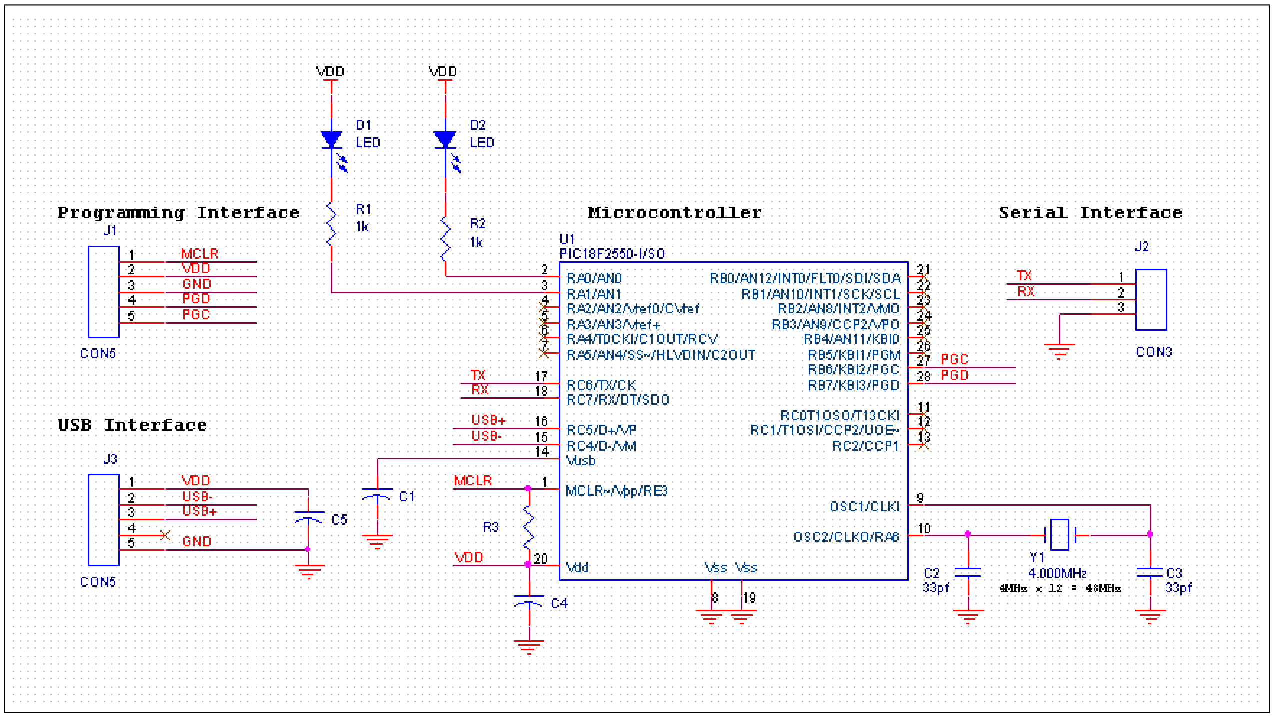

Figure 1: Schematic Diagram

truly random and cannot be guessed or predicted in any way.

There are several methods for generating random numbers in hardware and one of the most suited to this application is using the Avalanche effect of a semiconductor diode. The Avalanche effect is created by reverse-biasing an Avalanche or Zener diode. The noise generated is then amplified and passed into the analog-to-digital converter (ADC) of the microcontroller. The lowest significant bits (LSBs) are discarded to reduce the effects of nonlinearities in the ADC. The most significant bits (MSBs) are also discarded to ensure that the captured bits don't include extraneous zeros that are above the level of the noise.

The circuit shown in Figure 2 was tested to provide the random noise input to the microcontroller. The base to emitter junction of Q1 is used as the avalanche diode in this implementation. The final design did not include this random number generator because of the higher voltage requirements of this circuit to reach the avalanche region of the diode.

This circuit was prototyped on a perfboard as shown in Figure 3 and connected to the analog to digital converter on the microcontroller.

The source data collected from the tests were put through the Diehard tests[1] for random numbers and had excellent results as shown in Figure 4.

2.3.3 Printed Circuit Board (PCB)

The printed circuit board was produced by mechanical etching. The gerber files produced from the FreePCB software were imported into CopperCAM where the isolation routing data was created and exported as g-code. A computer numeric control (CNC) router was then used to isolation route a blank double-sided copper clad PCB.

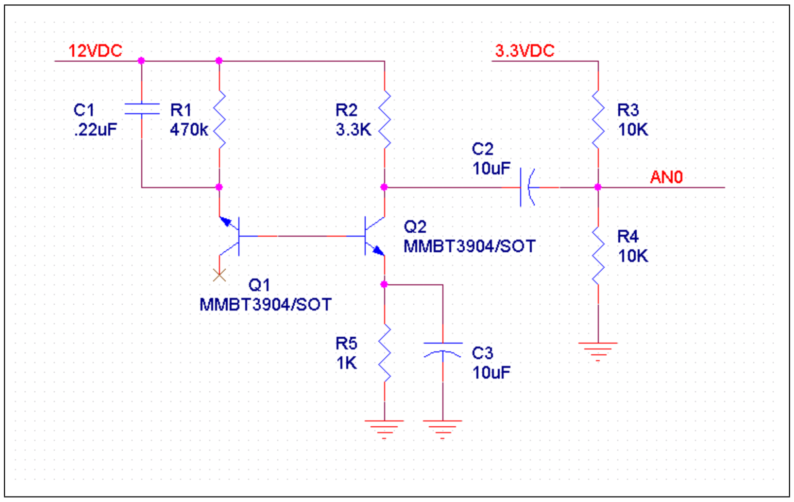

Figure 2: Schematic Diagram of Random Number Generator

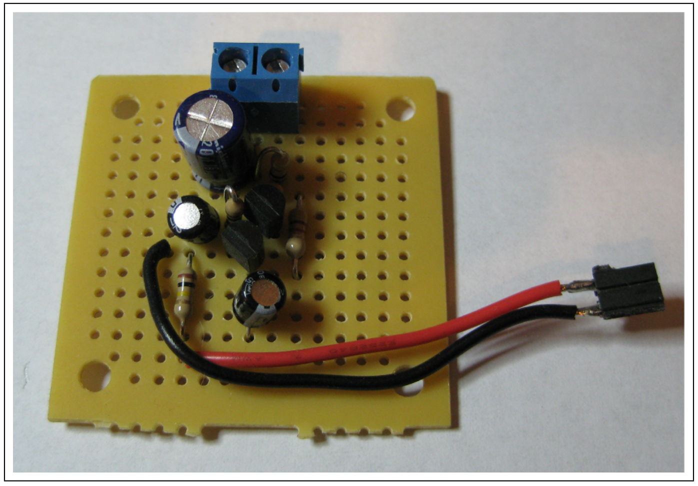

Figure 3: Prototype Random Number Generator

2.3.4 Assembly and Testing

The routed PCB was then hand assembled using standard techniques and is shown in Figure 5. The finished prototype was then tested for basic functionality and interfacing to a personal computer.

3 Software

The microcontroller software is written using a framework of C language with hand coded assembly language for most of the algorithms to improve efficiency. The code runs directly on the microcontroller without an operating system.

Test	Results
RGB Bit Distribution	PASSED at $> 5\%$
RGB Generalized Minimum Distance	PASSED at $> 5\%$
RGB Permutations	PASSED at $> 5\%$
RGB Lagged Sum*	PASSED at $> 5\%$
RGB Permutations	PASSED at $> 5\%$
Diehard Birthdays	PASSED at $> 5\%$
Diehard 32x32 Binary Rank	PASSED at $> 5\%$
Diehard 6x8 Binary Rank	PASSED at $> 5\%$
Diehard Bitstream	PASSED at $> 5\%$
Diehard OPSO	PASSED at $> 5\%$
Diehard OQSO	PASSED at $> 5\%$
Diehard DNA	PASSED at $> 5\%$
Diehard Count the 1s (stream)	PASSED at $> 5\%$
Diehard Count the 1s (byte)	PASSED at $> 5\%$
Diehard Parking Lot	PASSED at $> 5\%$
Diehard Minimum Distance (2d Circle)	PASSED at $> 5\%$
Diehard 3d Sphere (Minimum Distance)	PASSED at $> 5\%$
Diehard Squeeze	PASSED at $> 5\%$
Diehard Runs	PASSED at $> 5\%$
Diehard Craps	PASSED at $> 5\%$
Marsaglia and Tsang GCD	PASSED at $> 5\%$
STS Monobit	PASSED at $> 5\%$
STS Runs	PASSED at $> 5\%$
STS Serial	PASSED at $> 5\%$

* POSSIBLY WEAK on one run

Figure 4: Results of Diehard Tests

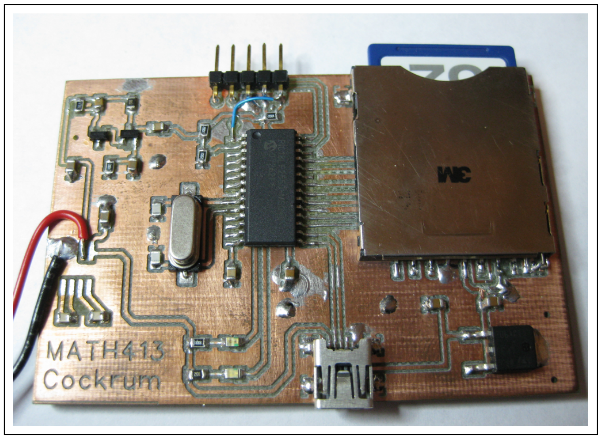

Figure 5: Photo of Completed Prototype Hardware

3.1 Algorithms Required

Since the word size is 8 bits, each 256 bit number is broken up into 8 bit chunks. The inputs to each algorithm are represented as A and B and are broken up into 32 8 bit bytes as follows:

$$A = \sum_{i=0}^{31} a_i 2^{8i} \tag{5}$$

$$B = \sum_{i=0}^{31} b_i 2^{8i} \tag{6}$$

3.1.1 Addition in a Prime Field

Addition is performed using the standard algorithm in 8 bit words using a hardware carry bit as shown below. The subsequent additions utilize an instruction (ADDWFC) to add the file register to the working register and add the carry bit in a single instruction cycle. This minimizes the cycles required to perform the 256 bit addition. If there is a final carry, the result is reduced by adding $r = 2^{256} = 2^{224} - 2^{192} - 2^{96} + 1$

```
Pseudocode:
carry=0
for i = 0 to 31
  c(i) = a(i) + b(i) + carry (Limited to 8 bits by ADDWFC command / memory width)
  if (a(i)+b(i)+carry) > 255 (Overflow bit from ADDWFC command)
    carry=1
  else
    carry=0
  endif
endfor

if carry=1
  add r  (see text)

C=A+B
```

3.1.2 Subtraction in a Prime Field

Subtraction is performed using the standard algorithm in 8 bit words using a hardware borrow bit. as shown below. The subsequent subtractions utilize an instruction (SUBWFB) to subtract the file register from the working register and subtract the borrow bit in a single instruction cycle. This minimizes the cycles required to perform the 256 bit subtraction. If there is an outstanding borrow, the output is represented as a two's complement negative number and p is added to put it in the interval $(0, p-1)$

```
Pseudocode:
borrow=0
for i = 0 to 31
  c(i) = a(i) - b(i) - borrow (Limited to 8 bits by SUBWFB command / memory width)
  if (a(i)-b(i)-borrow) < 0 (Overflow bit from SUBWFB command)
    borrow=1
  else
    borrow=0
  endif
endfor

if borrow=1
  add p using addition algorithm

C=A-B
```

3.1.3 Multiplication in a Prime Field

The PIC18F2550 microcontroller contains a hardware 8 bit x 8 bit multiplier that significantly reduces the number of cycles required to do a multiplication. To take advantage of this hardware multiplier, the standard long multiplication algorithm was used as follows:

```
Pseudocode:
for k = 0 to 31
  for n = 0 to 31
     PRODH:PRODL = a(n)*b(k) (PRODH:PRODL represents the concatenated 8 bit outputs of the multiply)
     c(n+k)=c(n+k) + PRODL (add with carry)
     c(n+k+1)=c(n+k+1) + PRODH (add with carry)
  endfor
endfor

C=A*B
```

3.1.4 Modular Reduction in a Prime Field

Since P-256 uses a generalized Mersenne prime modulus, fast methods for modular reduction exist [5]. The straight forward method for doing modulus reduction is to perform the standard division algorithm and retain the remainder. This is particularly slow on a low-power 8-bit microcontroller.

The algorithm shown by Solinas [5] and duplicated in FIPS-186 assumes a computer with a 32 bit bus width.

The generalized Mersenne prime is:

$$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \tag{7}$$

Let

$$B = A \mod p \tag{8}$$

Since A_i are 32 bits, every integer less than p^2 can be written as [5]:

$$\begin{aligned} A = A_{15} \cdot 2^{480} + A_{14} \cdot 2^{448} + A_{13} \cdot 2^{416} + A_{12} \cdot 2^{384} + A_{11} \cdot 2^{352} + \\ A_{10} \cdot 2^{320} + A_9 \cdot 2^{288} + A_8 \cdot 2^{256} + A_7 \cdot 2^{224} + A_6 \cdot 2^{192} + \\ A_5 \cdot 2^{160} + A_4 \cdot 2^{128} + A_3 \cdot 2^{96} + A_2 \cdot 2^{64} + A_1 \cdot 2^{32} + A_0 \end{aligned} \tag{9}$$

And since we are working with an 8-bit word size, for 8-bit a_i, every integer less than p^2 can be written as:

$$\begin{aligned} a = a_{63} \cdot 2^{504} + a_{62} \cdot 2^{496} + a_{61} \cdot 2^{488} + a_{60} \cdot 2^{480} + \cdots \\ \cdots + a_4 \cdot 2^{40} + a_3 \cdot 2^{32} + a_2 \cdot 2^{24} + a_1 \cdot 2^{16} + a_0 \end{aligned} \tag{10}$$

and the 256 bit term for A is:

$$A = a = (a_{63}||a_{62}|| \cdots ||a_1||a_0) \tag{11}$$

From Solinas [5], the expression for B is:

$$B = T + 2S_1 + 2S_2 + S_3 + S_4 - D_1 - D_2 - D_3 - D_4 \mod p \tag{12}$$

where the 256-bit terms are:

$$\begin{array}{rcccccccccccccccccc}
T = & (& A_7 & \| & A_6 & \| & A_5 & \| & A_4 & \| & A_3 & \| & A_2 & \| & A_1 & \| & A_0 &) \\
S_1 = & (& A_{15} & \| & A_{14} & \| & A_{13} & \| & A_{12} & \| & A_{11} & \| & 0 & \| & 0 & \| & 0 &) \\
S_2 = & (& 0 & \| & A_{15} & \| & A_{14} & \| & A_{13} & \| & A_{12} & \| & 0 & \| & 0 & \| & 0 &) \\
S_3 = & (& A_{15} & \| & A_{14} & \| & 0 & \| & 0 & \| & 0 & \| & A_{10} & \| & A_9 & \| & A_8 &) \\
S_4 = & (& A_8 & \| & A_{13} & \| & A_{15} & \| & A_{14} & \| & A_{13} & \| & A_{11} & \| & A_{10} & \| & A_9 &) \\
D_1 = & (& A_{10} & \| & A_8 & \| & 0 & \| & 0 & \| & 0 & \| & A_{13} & \| & A_{12} & \| & A_{11} &) \\
D_2 = & (& A_{11} & \| & A_9 & \| & 0 & \| & 0 & \| & A_{15} & \| & A_{14} & \| & A_{13} & \| & A_{12} &) \\
D_3 = & (& A_{12} & \| & 0 & \| & A_{10} & \| & A_9 & \| & A_8 & \| & A_{15} & \| & A_{14} & \| & A_{13} &) \\
D_4 = & (& A_{13} & \| & 0 & \| & A_{11} & \| & A_{10} & \| & A_9 & \| & 0 & \| & A_{15} & \| & A_{14} &)
\end{array} \tag{13}$$

and to get to 8-bit words, the following substitutions are made:

$$\begin{array}{rcccccccccc}
A_0 = & (& a_3 & \| & a_2 & \| & a_1 & \| & a_0 &) \\
A_1 = & (& a_7 & \| & a_6 & \| & a_5 & \| & a_4 &) \\
A_2 = & (& a_{11} & \| & a_{10} & \| & a_9 & \| & a_8 &) \\
A_3 = & (& a_{15} & \| & a_{14} & \| & a_{13} & \| & a_{12} &) \\
A_4 = & (& a_{19} & \| & a_{18} & \| & a_{17} & \| & a_{16} &) \\
A_5 = & (& a_{23} & \| & a_{22} & \| & a_{21} & \| & a_{20} &) \\
A_6 = & (& a_{27} & \| & a_{26} & \| & a_{25} & \| & a_{24} &) \\
A_7 = & (& a_{31} & \| & a_{30} & \| & a_{29} & \| & a_{28} &) \\
A_8 = & (& a_{35} & \| & a_{34} & \| & a_{33} & \| & a_{32} &) \\
A_9 = & (& a_{39} & \| & a_{38} & \| & a_{37} & \| & a_{36} &) \\
A_{10} = & (& a_{43} & \| & a_{42} & \| & a_{41} & \| & a_{40} &) \\
A_{11} = & (& a_{49} & \| & a_{46} & \| & a_{45} & \| & a_{44} &) \\
A_{12} = & (& a_{51} & \| & a_{50} & \| & a_{49} & \| & a_{48} &) \\
A_{13} = & (& a_{55} & \| & a_{54} & \| & a_{53} & \| & a_{52} &) \\
A_{14} = & (& a_{59} & \| & a_{58} & \| & a_{55} & \| & a_{56} &) \\
A_{15} = & (& a_{63} & \| & a_{62} & \| & a_{61} & \| & a_{60} &)
\end{array} \tag{14}$$

3.1.5 Inverse in a Prime Field

The inverse of a number in the field modulo p is calculated using the extended euclidean algorithm as follows:

```
Pseudocode:
u=a
v=p
x1=1
x2=0
while (u != 1) && (v != 1)
  while !(u & 1)                (while u is even)
    u=u>>1                      (divide by 2)
    if !(x1 & 1)                (if x1 is even)
      x1=x1>>1                  (divide by 2)
    else
      x1=(x1+p)>>1              (x1=(x1+p)/2 )
    endif
  endwhile
  while !(v & 1)                (while v is even)
```

```
    v=v>>1                        (divide by 2)
    if !(x2 & 1)                  (if x2 is even)
      x2=x2>>1                    (divide by 2)
    else
      x2=(x2+p)>>1                (x2=(x2+p)/2 )
    endif
  endwhile
  if (u > v)
    u=u-v
    x1=x1-x2
  else
    v=v-u
    x2=x2-x1
  endif
endwhile
if (u==1)
  inverse = x1
else
  inverse = x2
endif
```

3.2 Elliptic Curve Point Addition

Let $P = (P_x, P_y), Q = (Q_x, Q_y)$ be points on an elliptic curve over a prime field with neither equal to the point at infinity and $P \neq -Q$. Then the following rules are used to add two points using affine coordinates.

$\lambda = \frac{P_y - Q_y}{P_x - Q_x}$
$S = (S_x, S_y) = P + Q$
$S_x = \lambda^2 - P_x - Q_x$
$S_y = \lambda(Q_x - S_x) - Q_y$

3.3 Elliptic Curve Point Doubling

Let $P = (P_x, P_y)$ be a point on the NIST p256 elliptic curve with P not equal to the point at infinity. Then the following rules are used to double a point using affine coordinates.

$\lambda = \frac{3(Q_x^2 - 1)}{2Q_y}$
$D = (D_x, D_y) = 2P$
$D_x = \lambda^2 - P_x - Q_x$
$D_y = \lambda(Q_x - D_x) - Q_y$

3.4 Elliptic Curve Point Multiplication

Point multiplication isn't defined as straight forward like addition or doubling and the most straight forward algorithm to perform this operation uses an add and double method known also as the MSB binary method [2]. Let $P = (P_x, P_y)$ be a point on an elliptic curve with P not equal to the point at infinity and let k be an integer in the range $(2, r - 1)$ where r is the order of the curve. The algorithm follows:

```
Pseudocode:
Q= 0                            (point at infinity)
for i = 255 to 0
  Q=2Q                          (double)
  if k(i) == 1                  (bit i of k)
    then Q=Q+P                  (addition)
  endif
endfor

Q=k*P
```

4 Results

The efficiency of each of the algorithms in the prime field was measured by counting the cycles used on a simulator and then verifying the results by running in real hardware. The elliptic curve point addition, doubling, and multiplication results were calculated using the actual times from the prime field algorithms and the number of them required. Since the number of '1' bits affects the number of additions that must be performed, this was estimated to be one half of the total length (i.e. $256/2 = 128$ bits). The multiplication, modulus p, and inverse algorithms are significantly slower than the addition and subtraction algorithms as shown in Figure 6.

Algorithm	Cycles	Time
Addition	206	17.2 uS
Subtraction	273	22.75 uS
Multiplication	15803	1317 uS
Modulus p reduction	12790	1066 uS
Inverse	31280	2607 uS
Elliptic Curve Point Addition (1 Inv, 6 Sub, 2 Mul)	64524	5.4 mS
Elliptic Curve Point Doubling (1 Inv, 5 Sub, 4 Mul)	95857	8.0 mS
Elliptic Curve Point Multiplication (256 EC Dbl, 128 EC Add)	32798464	2.73 S

Figure 6: Algorithm Efficiency

Assuming that the communications time is negligible, the PIC18F2550 microcontroller can perform a Diffie-Hellman key exchange in approximately 5.4 seconds (2 elliptic curve point multiplications). The addition of the improvements listed in the next section should be able to significantly reduce this time.

The implementation of the prime field algorithms used 635 bytes of RAM (data) memory and 4072 bytes of ROM (program) memory. This accounts for approximately 31 percent of the RAM (data) memory and approximately 13 percent of the ROM (program) memory available on the microcontroller.

5 Possible Improvements

5.1 Elliptic Curve Coordinates

Although the use of affine coordinates is the most straight forward, the use of standard projective or Jacobian projective coordinates may significantly speed up the elliptic curve Diffie-Hellman algorithm. For example [2] , a point doubling using affine coordinates uses 1 inversion, 2 multiplications, and 2 squarings. The same operation using Jacobian projective coordinates uses 4 multiplications and 2 squarings. Since the computational cost of computing an inverse is significantly more than a multiplication, the Jacobian projective coordinates is faster for this operation. Further research is required to determine which mix of coordinate systems is most efficient for this application.

5.2 Dedicated Squaring Algorithm

In this implementation, squaring is performed using a multiplication algorithm. The availability of a hardware 8x8 bit multiplier makes the multiplication significantly faster and more efficient than on microcontrollers that don't have this hardware. A performance analysis of the hardware multiplier versus software squaring may uncover possible performance gains.

5.3 Modular Reduction Algorithm Improvement

The fast modular reduction shown in Solinas's paper [5] was calculated for a 32 bit word size machine. This implementation uses a substitution of four 8 bit words into the algorithm. Additional performance gains may be uncovered by deriving the algorithm for an 8 bit word size.

5.4 Modular Reduction Coding Improvement

Currently, the modular reduction algorithm constructs each temporary variable $(T, S_1, S_2, \cdots D_3, D_4)$ in its entirety then calls the addition or subtraction algorithm. Since many of the words that compose these temporary variables are zero, the algorithm may be coded to only add or subtract the non-zero values which would result in a speed improvement while possibly increasing the memory size.

5.5 Speed versus Memory Trade offs

This implementation can be made faster by unrolling all of the loops in the software to eliminate the counts and compares used for the looping operation. Conversely, it could also be made smaller (more memory efficient) by using recursion and additional looping. This trade off should be considered in future implementations.

5.6 Code Reorganization

Most of the functions implemented have a separate output variable so that the operation can be performed in a non-destructive way. Reorganizing to allow in-place operations will eliminate the copies and most of the zeroing that is done at the beginning of each function. This will speed up most of the field operations by over 100 cycles.

5.7 Assembly Coding

Most of the computationally intensive sections of this implementation have been coded in assembly language which reduces a number of inefficiencies injected by a C compiler. Additional speed and memory efficiency may be gained by hand coding the entire implementation. This is a trade off with available time and readability versus possibly negligible performance gains.

6 Conclusion

The PIC18F2550 microcontroller is easily capable of performing an elliptic curve Diffie-Hellman key exchange. With an working time of 5.4 seconds per exchange, this type of cryptography is not suited to high speed data transfers on this device. For high-speed transfers, the exchanged secret may be used as the key in a symmetric cipher such as Rijndael (as used in the Advanced Encryption Standard [4]).

7 Acknowledgements

This report was prepared as a final project for Math 413 Number Theory at University of Maryland, Baltimore County. Thanks to Robert Campbell (rcampbel@umbc.edu) for his support in this course and on this paper.

References

[1] Robert G. Brown. Dieharder: A Random Number Test Suite. http://www.phy.duke.edu/ rgb/General/dieharder.php. [2009 May 7].

[2] S.S. Kumar. *Elliptic Curve Cryptography for Constrained Devices*. Bochum, 2008.

[3] U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology. FIPS 186-2: Digital Signature Standard (DSS). http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf. [2000 January 27].

[4] U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology. FIPS 197: Advanced Encryption Standard (AES). http://www.techheap.com/cryptography/encryption/fips-197.pdf. [2001 November 26].

[5] J.A. Solinas, Faculty of Mathematics, Dept. of Combinatorics, Optimization, and University of Waterloo. *Generalized mersenne numbers*. Faculty of Mathematics, University of Waterloo, 1999.

8 Appendix A. Source Code for Algorithms

```
/*******************************************/
/* Large Integer Math Library              */
/* for arithmetic in the prime field       */
/* modulus = 2^256 - 2^224 + 2^192 + 2^96 -1 */
/* by Chris K Cockrum, Spring 2009         */
/* CPU: Microchip PIC18F2550               */
/*******************************************/
#include <18F2550.h>

#fuses HSPLL,NOWDT,NOPROTECT,NOLVP,NODEBUG,USBDIV,PLL1,CPUDIV2,VREGEN
#DEVICE ADC=10

#use delay(clock=48000000)

/* Define additional registers needed for ASM code */
#define STATUS    0x0FD8
#define FSR0L     0x0FE9
#define FSR0H     0x0FEA
#define FSR1L     0x0FE1
#define FSR1H     0x0FE2
#define FSR2L     0x0FD9
#define FSR2H     0x0FDA
#define INDF0     0x0FEF
#define INDF1     0x0FE7
#define INDF2     0x0FDF
#define POSTINC0  0x0FEE
#define POSTINC1  0x0FE6
#define POSTINC2  0x0FDE
#define POSTDEC0  0x0FED
#define POSTDEC1  0x0FE5
#define POSTDEC2  0x0FDD
#define PREINC0   0x0FEC
#define PREINC1   0x0FE4
#define PREINC2   0x0FDC
#define PRODH     0x0FF4
#define PRODL     0x0FF3

/* Format of large integers */
/* 0x010203 ... 32 = {0x32, 0x31, 0x29, ... 0x01} */
/* This is done to easily increment pointers starting with the LSB */

/* Define my secret number */
/* This should be randomly generated it's easier to demonstrate with a fixed number */
byte secret[32] = { 0xDE,0xAD,0xBE,0xEF,0xDE,0xAD,0xBE,0xEF,
                    0xAB,0xAD,0xBA,0xBE,0xAB,0xAD,0xBA,0xBE,
                    0xDE,0xAD,0xBE,0xEF,0xDE,0xAD,0xBE,0xEF,
                    0xAB,0xAD,0xBA,0xBE,0xAB,0xAD,0xBA,0xBE };

/* Using NIST prime field p256 = 2^256 - 2^224 + 2^192 + 2^96 -1 */

/* Define Base Point on Elliptic Curve */
```

```
byte Gx[32]={ 0x96, 0xc2, 0x98, 0xd8, 0x45, 0x39, 0xa1, 0xf4,
              0xa0, 0x33, 0xeb, 0x2d, 0x81, 0x7d, 0x03, 0x77,
              0xf2, 0x40, 0xa4, 0x63, 0xe5, 0xe6, 0xbc, 0xf8,
              0x47, 0x42, 0x2c, 0xe1, 0xf2, 0xd1, 0x17, 0x6b };

byte Gy[32]={ 0xf5, 0x51, 0xbf, 0x37, 0x68, 0x40, 0xb6, 0xcb,
              0xce, 0x5e, 0x31, 0x6b, 0x57, 0x33, 0xce, 0x2b,
              0x16, 0x9e, 0x0f, 0x7c, 0x4a, 0xeb, 0xe7, 0x8e,
              0x9b, 0x7f, 0x1a, 0xfe, 0xe2, 0x42, 0xe3, 0x4f };

byte x[32] = { 0x11,0x02,0x03,0x04,0x05,0x06,0x07,0x08,
               0x11,0x02,0x03,0x04,0x05,0x06,0x07,0x08,
               0x11,0x02,0x03,0x04,0x05,0x06,0x07,0x08,
               0x11,0x02,0x03,0x04,0x05,0x06,0x07,0xf8};

byte y[32] = { 0x01,0xC2,0x03,0x04,0x05,0x06,0x07,0x08,
               0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,
               0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,
               0x01,0x02,0x03,0x04,0x05,0x06,0x07,0xF8};

/* Global Variables */
byte z[32];       /* For Testing */
byte bz[64];      /* For Testing */
byte tempm[32];   /* Temp for mod p and invert */
byte tempm2[32];  /* Temp for mod p and invert */
byte tempas[32];  /* Temp for add/subtract */
byte r[32];
byte s[32];

long time;

byte u[32],v[32],x1[32],x2[32];   /* For Binary Extended Euclidean Algorithm (lip_inv) */

/* This is added after a subtract if the answer is negative (2's complement */
static byte modulus[32]={  0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
                           0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x00,
                           0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
                           0x01,0x00,0x00,0x00,0xff,0xff,0xff,0xff };

/* This is added after an addition if the answer has a carry */
static byte reducer[32]={  0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
                           0x00,0x00,0x00,0x00,0xff,0xff,0xff,0xff,
                           0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
                           0xfe,0xff,0xff,0xff,0x00,0x00,0x00,0x00};

/* Function prototypes */
void lip_mul(byte *A, byte *B, byte *C);
void lip_inv(byte *A, byte *B);
void lip_copy(byte *A, byte *B);
void lip_modp(byte *A, byte *B);
void lip_add(byte *A, byte *B, byte *C);
void lip_sub(byte *A, byte *B, byte *C);

/* Internal Functions */
```

```
int lip_addt(byte *A, byte *B, byte *C);
int lip_subt(byte *A, byte *B, byte *C);
void lip_rshift(byte *A);
int lip_notzero(byte *A);

/*******************************/
/* Function Name: lip_add       */
/*    C = A + B                 */
/*    A,B,C = 32 byte arrays    */
/*    Return: N/A               */
/*******************************/
void lip_add(byte *A, byte *B, byte *C)
{
   if(lip_addt(A,B,C))
   {
      lip_addt(C,reducer,tempas);
      lip_copy(tempas,C);
   }
}

/*******************************/
/* Function Name: lip_sub       */
/*    C = A - B                 */
/*    A,B,C = 32 byte arrays    */
/*    Return: N/A               */
/*******************************/
void lip_sub(byte *A, byte *B, byte *C)
{
   if(lip_subt(A,B,C))
   {
      lip_addt(C,modulus,tempas);
      lip_copy(tempas,C);
   }
}

/*******************************/
/* Function Name: lip_modp      */
/*    A = B mod modulus         */
/*    A = 32 byte array         */
/*    B = 64 byte array         */
/*    Return: N/A               */
/*******************************/
void lip_modp(byte *A, byte *B)
{
   BYTE n;

   /* A = T */
   lip_copy(B,A);

   /* Form S1 */
   for(n=0;n<12;n++)
      tempm[n]=0;
   for(n=12;n<32;n++)
      tempm[n]=B[n+32];
```

```
/* tempm2=T+S1 */
lip_add(A,tempm,tempm2);

/* A=T+S1+S1 */
lip_add(tempm2,tempm,A);

/* Form S2 */
for(n=0;n<12;n++)
   tempm[n]=0;
for(n=12;n<28;n++)
   tempm[n]=B[n+36];
for(n=28;n<32;n++)
   tempm[n]=0;

/* tempm2=T+S1+S1+S2 */
lip_add(A,tempm,tempm2);

/* A=T+S1+S1+S2+S2 */
lip_add(tempm2,tempm,A);

/* Form S3 */
for(n=0;n<12;n++)
   tempm[n]=B[n+32];
for(n=12;n<24;n++)
   tempm[n]=0;
for(n=24;n<32;n++)
   tempm[n]=B[n+32];

/* tempm2=T+S1+S1+S2++S2+S3 */
lip_add(A,tempm,tempm2);

/* Form S4 */
for(n=0;n<12;n++)
   tempm[n]=B[n+36];
for(n=12;n<24;n++)
   tempm[n]=B[n+40];
for(n=24;n<28;n++)
   tempm[n]=B[n+28];
for(n=28;n<32;n++)
   tempm[n]=B[n+4];

/* A=T+S1+S1+S2++S2+S3+S4 */
lip_add(tempm2,tempm,A);

/* Form D1 */
for(n=0;n<12;n++)
   tempm[n]=B[n+44];
for(n=12;n<24;n++)
   tempm[n]=0;
for(n=24;n<28;n++)
   tempm[n]=B[n+8];
for(n=28;n<32;n++)
   tempm[n]=B[n+12];
   /* tempm2=T+S1+S1+S2++S2+S3+S4-D1 */
   lip_sub(A,tempm,tempm2);

   /* Form D2 */
   for(n=0;n<16;n++)
      tempm[n]=B[n+48];
   for(n=16;n<24;n++)
      tempm[n]=0;
   for(n=24;n<28;n++)
      tempm[n]=B[n+12];
   for(n=28;n<32;n++)
      tempm[n]=B[n+16];

   /* A=T+S1+S1+S2++S2+S3+S4-D1-D2 */
   lip_sub(tempm2,tempm,A);

   /* Form D3 */
   for(n=0;n<12;n++)
      tempm[n]=B[n+52];
   for(n=12;n<24;n++)
      tempm[n]=B[n+20];
   for(n=24;n<28;n++)
      tempm[n]=0;
   for(n=28;n<32;n++)
      tempm[n]=B[n+20];

   /* tempm2=T+S1+S1+S2++S2+S3+S4-D1-D2-D3 */
   lip_sub(A,tempm,tempm2);

   /* Form D4 */
   for(n=0;n<8;n++)
      tempm[n]=B[n+56];
   for(n=8;n<12;n++)
      tempm[n]=0;
   for(n=12;n<24;n++)
      tempm[n]=B[n+24];
   for(n=24;n<28;n++)
      tempm[n]=0;
   for(n=28;n<32;n++)
      tempm[n]=B[n+24];

   /* A=T+S1+S1+S2++S2+S3+S4-D1-D2-D3-D4 */
   lip_sub(tempm2,tempm,A);
}

/*******************************/
/* Function Name: lip_mul      */
/*    C = A x B                */
/*    A,B = 32 byte arrays     */
/*    C = 64 byte array        */
/*    Return: N/A              */
/*******************************/
void lip_mul(byte *A, byte *B, byte *C)
```

```
{
   BYTE shift=0;           /* Keeps track of digit in output */
   BYTE out_loop;          /* Main loop counter */
   BYTE loop;              /* Loop variable */
   BYTE carryL;            /* CarryL variable */
   BYTE carryH;            /* CarryH variable */

#asm

; Clear output variable = C
movff C, FSR1L
   movff &C+1, FSR1H
   movlw 64               ; Copy 64 to w
   movwf loop             ; Copy w to loop
loopClear:
   clrf POSTINC1          ; Clear C(n)
   decfsz loop            ; loop test
   bra loopClear          ; jump

   movlw 31               ; Set outer loop counter to 31
   movwf out_loop

; Set up pointers
   movff A, FSR0L
   movff &A+1, FSR0H
   movff B, FSR1L
   movff &B+1, FSR1H
   movff C, FSR2L
   movff &C+1, FSR2H

   movlw 31               ; Set loop counter to 31
   movwf loop

   movf POSTINC0,w        ; Move A to w
   mulwf INDF1            ; PRODH:PRODL = A(0) * B(0)
   movff PRODL, POSTINC2  ; C(0)=PRODL
   movff PRODH, INDF2     ; C(1)=PRODL

loop01:
   movf POSTINC0, w       ; w=A(i)
   mulwf INDF1            ; PRODH:PRODL = A(i)* B(n)
   movf PRODL, w          ; w=PRODL
   addwf POSTINC2         ; C(n)=C(n)+w
   movf PRODH, w          ; w=PRODH
   addwfc INDF2           ; C(n)=C(n)+w+carry flag
   decfsz loop            ; loop test
   bra loop01             ; jump
   movlw 31               ; Set outer loop counter to 31
   movwf out_loop
   clrf carryH
   movff B, FSR1L
   movff &B+1, FSR1H

loopOuter:
   movff A, FSR0L
   movff &A+1, FSR0H
   movff C, FSR2L
   movff &C+1, FSR2H

   ; Shift index of c by shift
   incf shift              ; Increment shift
   movf shift,w            ; Copy shift to w
   addwf FSR2L, 1          ; Add shift to FSR2L
   movlw 0                 ; Zero the w register
   addwfc FSR2H, 1         ; Add carry to FSR2H
   movf POSTINC0,w         ; Move A to w
   mulwf PREINC1           ; PRODH:PRODL = A(i) * B(n)  and ++n
   movf PRODL,w            ; w=PRODL
   addwf POSTINC2          ; C(k)=C(k)+PRODL and k++Memory vs Speed Tradeoffs
   movf PRODH, w           ; w=PRODH
   addwfc INDF2            ; C(k)=C(k)+PRODH+carryflag
   clrf carryH             ; carryH=0
   btfsc STATUS,0          ; test carry flag
   bsf carryH,0            ; carryH=1
   movlw 31                ; Set loop counter to 31
   movwf loop

loop02:
   movf POSTINC0,w         ; Move A to w
   mulwf INDF1             ; PRODH:PRODL = A(i) * B(n)
   movf PRODL,w            ; w=PRODL
   addwf POSTINC2          ; C(k)=C(k)+PRODL and k++
   movf PRODH, w           ; w=PRODH
   addwfc INDF2            ; C(k)=C(k)+PRODH+carryflag
   clrf carryH             ; carryH=0
   btfsc STATUS,0          ; test for carryflag
   incf carryH             ; carryH++
   addwf INDF2             ; C(k)=C(k)+carryH
   btfsc STATUS,0          ; test for carryflag
   incf carryH             ; carryH++
   decfsz loop             ; loop test
   bra loop02              ; jump
   decfsz out_loop         ; loop test
   bra loopOuter           ; jump
#endasm
}

/********************************/
/* Function Name: lip_subt      */
/*    C = A - B                 */
/*    A,B,C = 32 byte arrays    */
/*    Return: borrow            */
/********************************/
int lip_subt(byte *A, byte *B, byte *C)
{
BYTE loop;
#asm
   movff A, FSR0L          ; Copy addresses to FSRs for indirect access
   movff &A+1, FSR0H
   movff B, FSR1L
   movff &B+1, FSR1H
   movff C, FSR2L
   movff &C+1, FSR2H

   movlw 31                ; Copy 32 to w
   movwf loop              ; Copy w to loop
   movff POSTINC0,INDF2    ; Copy B to C since adds are done in-place
   movf POSTINC1,0         ; Copy A to W
   subwf POSTINC2          ; C = C-w

loopSub:
   movff POSTINC0,INDF2    ; Copy B to C since adds are done in-place
   movf POSTINC1,0         ; w=B
   subwfb POSTINC2         ; C = C-w-c
   decfsz loop             ; loop test
   bra loopSub             ; jump

   btfsc STATUS.0          ; test borrow
   bra noborrow
   nop
   movlw 0x01
   movwf loop
noborrow:
#endasm

   return loop;
}

/*******************************/
/* Function Name: lip_rshift    */
/*    A  A >> 1                 */
/*    A = 32 byte array         */
/*    Return: N/A               */
/*******************************/
void lip_rshift(byte *A)
{
   BYTE loop;
   BYTE *temp;

   temp=A+31;

#asm
   movff temp, FSR0L       ; Copy address to FSR for indirect access
   movff &temp+1, FSR0H

   movlw 32                ; Copy 32 to w
   movwf loop              ; Copy w to loop
   bcf STATUS,0            ; Clear carry flag

loopRot:
   rrcf POSTDEC0
   decfsz loop             ; loop test
   bra loopRot             ; jump
#endasm
}

/*******************************/
/* Function Name: lip_notzero   */
/*    A = 32 byte arrays        */
/*    Return: 1 if not zero     */
/*******************************/
int lip_notzero(byte *A)
{
   byte n;
   for(n=0;n<32;n++)
      if (A[n]!=0)
         break;

   if (n==32)
      return 0;
   else
      return 1;
}

/*******************************/
/* Function Name: lip_isone     */
/*    A = 32 byte arrays        */
/*    Return: 1 if value is one */
/*******************************/
int lip_isone(byte *A)
{
   byte n;
   for(n=1;n<32;n++)
      if (A[n]!=0)
         break;

   if ((n==32)&&(A[0]==1))
      return 1;
   else
      return 0;
}

/*******************************/
/* Function Name: lip_inv       */
/*    B = A^-1                  */
/*    A,B = 32 byte arrays      */
/*    Return: N/A               */
/* Binary Euclidean Algorithm   */
/*******************************/
void lip_inv(byte *A, byte *B)
{
   byte n,t;

   lip_copy(A,u);
   lip_copy(modulus,v);
   for(n=0;n<32;n++)
   {
      x1[n]=0;
      x2[n]=0;
   }
   x1[0]=1;
   /* While u !=1 and v !=1 */
   while ((lip_isone(u) || lip_isone(v))==0)
   {
      while(!(u[0]&1))                  /* While u is even */
      {
         lip_rshift(u);                 /* divide by 2 */
         if (!(x1[0]&1))                /* if x1 is even */
            lip_rshift(x1);             /* Divide by 2 */
         else
         {
            lip_add(x1,modulus,tempm); /* tempm=x1+p */
            lip_copy(tempm,x1);        /* x1=tempm */
            lip_rshift(x1);             /* Divide by 2 */
         }
      }
      while(!(v[0]&1))                  /* While v is even */
      {
         lip_rshift(v);                 /* divide by 2 */
         if (!(x2[0]&1))                /* if x1 is even */
            lip_rshift(x2);             /* Divide by 2 */
         else
         {
            lip_add(x2,modulus,tempm); /* tempm=x1+p */
            lip_copy(tempm,x2);        /* x1=tempm */
            lip_rshift(x2);             /* Divide by 2 */
         }
      }
      t=lip_subt(u,v,tempm);            /* tempm=u-v */
      if (t==0)                         /* If u > 0 */
      {
         lip_copy(tempm,u);             /* u=u-v */
         lip_sub(x1,x2,tempm);          /* tempm=x1-x2 */
         lip_copy(tempm,x1);            /* x1=x1-x2 */
      }
      else
      {
         lip_subt(v,u,tempm);           /* tempm=v-u */
         lip_copy(tempm,v);             /* u=u-v */
         lip_sub(x2,x1,tempm);          /* tempm=x2-x1 */
         lip_copy(tempm,x2);            /* x2=x2-x1 */
      }
   }
   if (lip_isone(u))
      lip_copy(x1,B);
   else
      lip_copy(x2,B);
}

/*******************************/
```

```
/* Function Name: lip_copy      */
/*    B = A                     */
/*    A,B = 32 byte arrays      */
/*    Return: N/A               */
/*******************************/
void lip_copy(byte *A, byte *B)
{
BYTE loop;

#asm
   movff A, FSR0L   ; Copy address to FSR for indirect access
   movff &A+1, FSR0H
   movff B, FSR1L
   movff &B+1, FSR1H

   movlw 32                ; Copy 32 to w
   movwf loop              ; Copy w to loop
loopCopy:
   movff POSTINC0,POSTINC1 ; Copy B to A
   decfsz loop             ; loop test
   bra loopCopy            ; jump
#endasm
}

/*******************************/
/* Function Name: lip_addt      */
/*    C = A + B                 */
/*    A,B,C = 32 byte arrays    */
/*    Return: carry             */
/*******************************/
int lip_addt(byte *A, byte *B, byte *C)
{
BYTE loop;
#asm
   movff A, FSR0L  ; Copy address to FSR for indirect access
   movff &A+1, FSR0H
   movff B, FSR1L
   movff &B+1, FSR1H
   movff C, FSR2L
   movff &C+1, FSR2H

   movlw 31                ; Copy 32 to w
   movwf loop              ; Copy w to loop
   movff POSTINC1,INDF2    ; Copy B to C since adds are done in-place
   movf POSTINC0,0         ; Copy A to W
   addwf POSTINC2          ; C = W+C

loopAdd:
   movff POSTINC1,INDF2    ; Copy B to C since adds are done in-place
   movf POSTINC0,0         ; w=A
   addwfc POSTINC2         ; C=C+w+c
   decfsz loop             ; loop test
   bra loopAdd             ; jump
   btfss STATUS.0  ; test carry
   bra nocarry
   nop
   movlw 0x01
   movwf loop
nocarry:
#endasm
   return loop;
}

void main(void)
{
   byte n;

  /* Set up timer to increment every instruction cycle */
  /* At 48MHz this is at 12Mhz so period=.083333uS */
  setup_timer_1(T1_INTERNAL);

  /* Set Timer to 0 */
  set_timer1(0);

  lip_add(x,x,y);
  lip_sub(x,x,y);
  lip_mul(x,x,bz);
  lip_modp(z,bz);
  lip_inv(x,z);

  /* Get Timer Value */
  time=get_timer1();

   /* Wait here to keep debugger active */
   while(1);
}
```