Pseudo-Random Numbers for PLCs

Kurt Nalty

June 10, 2013

Abstract

Linear congruential generators (LCGs) are an easy, low overhead method of making pseudo-random numbers. Programmable Logic Controllers (PLCs) can benefit from LCGs when testing software, and when a randomizing response is needed. This paper discusses the characteristics and limitations of LCGs, and shows two implementations, one being Siemens SIMATIC S7-200 and the other being the Allen Bradley MicroLogix PLCs.

LCG Formulas

Linear congruential generators are based upon a simple linear equation, carried out using modulo arithmetic. For a sequence of values, with current value x_i, the next value in the sequence x_{i+1} is given by

$$x_{i+1} = (A * x_i + B) \bmod N$$

When we work with computers and PLCs on a byte basis, we automatically have mod 256 arithmetic. If we wish our sequence to cover all 256 possible byte values without duplication, we need $A = 4k + 1$ and $B = 2m + 1$. In other words, A must be one above a multiple of four, while B must be an odd number.

What follows is a hex sequence dump of the simplest LCG, with $A = 1$ and $B = 1$. You notice this is just the simple counting sequence.

```
 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
```

Next, look at the sequence where $A = 1$ and $B = 3$. Again, this is trivial, simply counting by threes and going around the corner when we hit 255.

```
 0  3  6  9  c  f 12 15 18 1b 1e 21 24 27 2a 2d
30 33 36 39 3c 3f 42 45 48 4b 4e 51 54 57 5a 5d
60 63 66 69 6c 6f 72 75 78 7b 7e 81 84 87 8a 8d
90 93 96 99 9c 9f a2 a5 a8 ab ae b1 b4 b7 ba bd
c0 c3 c6 c9 cc cf d2 d5 d8 db de e1 e4 e7 ea ed
f0 f3 f6 f9 fc ff  2  5  8  b  e 11 14 17 1a 1d
20 23 26 29 2c 2f 32 35 38 3b 3e 41 44 47 4a 4d
50 53 56 59 5c 5f 62 65 68 6b 6e 71 74 77 7a 7d
80 83 86 89 8c 8f 92 95 98 9b 9e a1 a4 a7 aa ad
b0 b3 b6 b9 bc bf c2 c5 c8 cb ce d1 d4 d7 da dd
e0 e3 e6 e9 ec ef f2 f5 f8 fb fe  1  4  7  a  d
10 13 16 19 1c 1f 22 25 28 2b 2e 31 34 37 3a 3d
40 43 46 49 4c 4f 52 55 58 5b 5e 61 64 67 6a 6d
70 73 76 79 7c 7f 82 85 88 8b 8e 91 94 97 9a 9d
a0 a3 a6 a9 ac af b2 b5 b8 bb be c1 c4 c7 ca cd
d0 d3 d6 d9 dc df e2 e5 e8 eb ee f1 f4 f7 fa fd
```

Now we look at the sequence where $A = 1$ and $B = 5$. Once again, the process is simply counting by fives.

```
 0  5  a  f 14 19 1e 23 28 2d 32 37 3c 41 46 4b
50 55 5a 5f 64 69 6e 73 78 7d 82 87 8c 91 96 9b
a0 a5 aa af b4 b9 be c3 c8 cd d2 d7 dc e1 e6 eb
f0 f5 fa ff  4  9  e 13 18 1d 22 27 2c 31 36 3b
40 45 4a 4f 54 59 5e 63 68 6d 72 77 7c 81 86 8b
90 95 9a 9f a4 a9 ae b3 b8 bd c2 c7 cc d1 d6 db
e0 e5 ea ef f4 f9 fe  3  8  d 12 17 1c 21 26 2b
30 35 3a 3f 44 49 4e 53 58 5d 62 67 6c 71 76 7b
80 85 8a 8f 94 99 9e a3 a8 ad b2 b7 bc c1 c6 cb
d0 d5 da df e4 e9 ee f3 f8 fd  2  7  c 11 16 1b
20 25 2a 2f 34 39 3e 43 48 4d 52 57 5c 61 66 6b
70 75 7a 7f 84 89 8e 93 98 9d a2 a7 ac b1 b6 bb
c0 c5 ca cf d4 d9 de e3 e8 ed f2 f7 fc  1  6  b
10 15 1a 1f 24 29 2e 33 38 3d 42 47 4c 51 56 5b
60 65 6a 6f 74 79 7e 83 88 8d 92 97 9c a1 a6 ab
b0 b5 ba bf c4 c9 ce d3 d8 dd e2 e7 ec f1 f6 fb
```

There are 8192 different byte size LCGs, which can be seen graphically as an animated gif at `http://www.kurtnalty.com/Animated_LCG.gif`. Each frame in this image is made from 128 sequences, each shown as a vertical black and white bitmap 8 bits wide by 256 bits down. Looking at the animated frames, you can clearly see various periodicities among some of the patterns.

Limitations of LCGs

The most important limitations of the LCGs, are the periodicities in the low order bits. Figure 1 is a graphical representation of the simple counting sequence (b0), each byte shown vertically with LSB at top, running top to bottom for four bytes, then advancing in a similar fashion 64 more times. The various periodicities are obvious for the counting sequence.

Going back to the previous text displays, we see that the low order nibbles are the same in each vertical column. This means that the bottom four bits

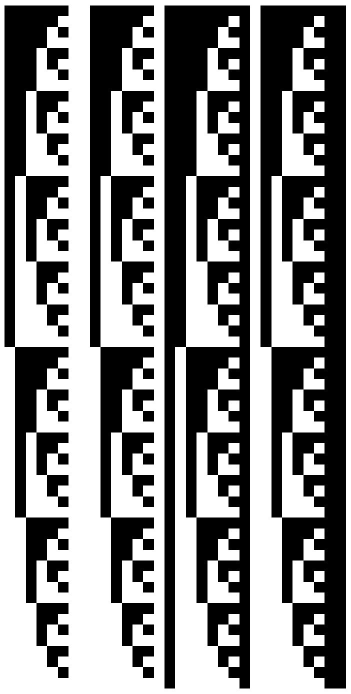


Figure 1: Counting Sequence (b0)

are periodic over sixteen samples.

Here is a later sequence, illustrating the same behavior.

```
 0 b1 ba 5b d4 65 4e cf 28 99 62 c3 fc 4d f6 37
50 81  a 2b 24 35 9e 9f 78 69 b2 93 4c 1d 46  7
a0 51 5a fb 74  5 ee 6f c8 39  2 63 9c ed 96 d7
f0 21 aa cb c4 d5 3e 3f 18  9 52 33 ec bd e6 a7
40 f1 fa 9b 14 a5 8e  f 68 d9 a2  3 3c 8d 36 77
90 c1 4a 6b 64 75 de df b8 a9 f2 d3 8c 5d 86 47
e0 91 9a 3b b4 45 2e af  8 79 42 a3 dc 2d d6 17
30 61 ea  b  4 15 7e 7f 58 49 92 73 2c fd 26 e7
80 31 3a db 54 e5 ce 4f a8 19 e2 43 7c cd 76 b7
d0  1 8a ab a4 b5 1e 1f f8 e9 32 13 cc 9d c6 87
20 d1 da 7b f4 85 6e ef 48 b9 82 e3 1c 6d 16 57
70 a1 2a 4b 44 55 be bf 98 89 d2 b3 6c 3d 66 27
c0 71 7a 1b 94 25  e 8f e8 59 22 83 bc  d b6 f7
10 41 ca eb e4 f5 5e 5f 38 29 72 53  c dd  6 c7
60 11 1a bb 34 c5 ae 2f 88 f9 c2 23 5c ad 56 97
b0 e1 6a 8b 84 95 fe ff d8 c9 12 f3 ac 7d a6 67
```

We find that the least significant bit toggles every count (period of two), the next bit has a period of four, the next bit has a period of eight, and so on and so forth. These bottom two bits just are not random enough, so I recommend - don't use them!

Here are a few more views of individual sequences, showing different scales of periodicity and randomness. You can see that the middle bits of these later sequences look pretty random.

Implementing LCGs on Siemens SIMATIC S7-200

While the S7-200 family has a built in multiply command, some PLCs don't. By choosing our factors wisely, we can use shift and add to achieve fairly fast multiplication. In this case, I chose $9x + 9 = 9(x + 1)$ for the LCG, simply because $9 = 8 + 1$, and multplying by eight can be done by shifting the bit pattern three bits to the left.

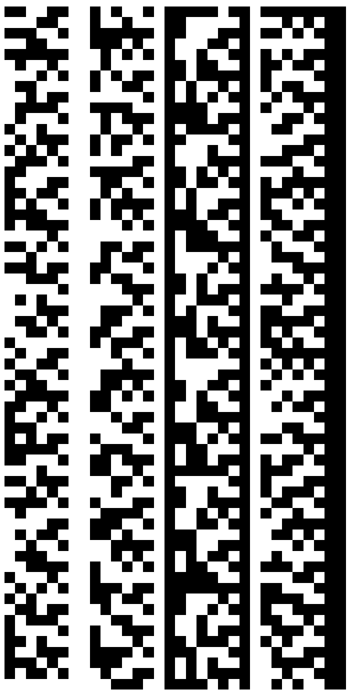


Figure 2: $9x_i + 9$ (b260)

Figure 3: (b4736)

This demonstration uses edge triggering off of a 1Hz system clock to call the subroutine where the LCG is carried out. Figure 4 shows the main program in ladder logic format. We use edge detection of the system 1Hz clock to only call our subroutine once per second. We then copy the bits 2-5 of the LCG value to our output lights, for this implementation.

Figure 5 shows the bitshifting based LCG routine using an STL assembly language view.

Of course, having a multiply command allows us to directly implement the LCG. A different implementation for the $185x_i + 21$ LCG is shown in Figure 6, using ladder logic format.

Implementing LCGs on Allen Bradley MicroLogix

The MicroLogix controllers don't have a one second clock as Siemens. However, they have a free running 10 kHz clock, and we can grab a bit from this clock for periodic updating.

Figure 7 shows the main program, very similar to our previous example. Figure 8 show the LCG subroutine, in this case for $89x + 35$.

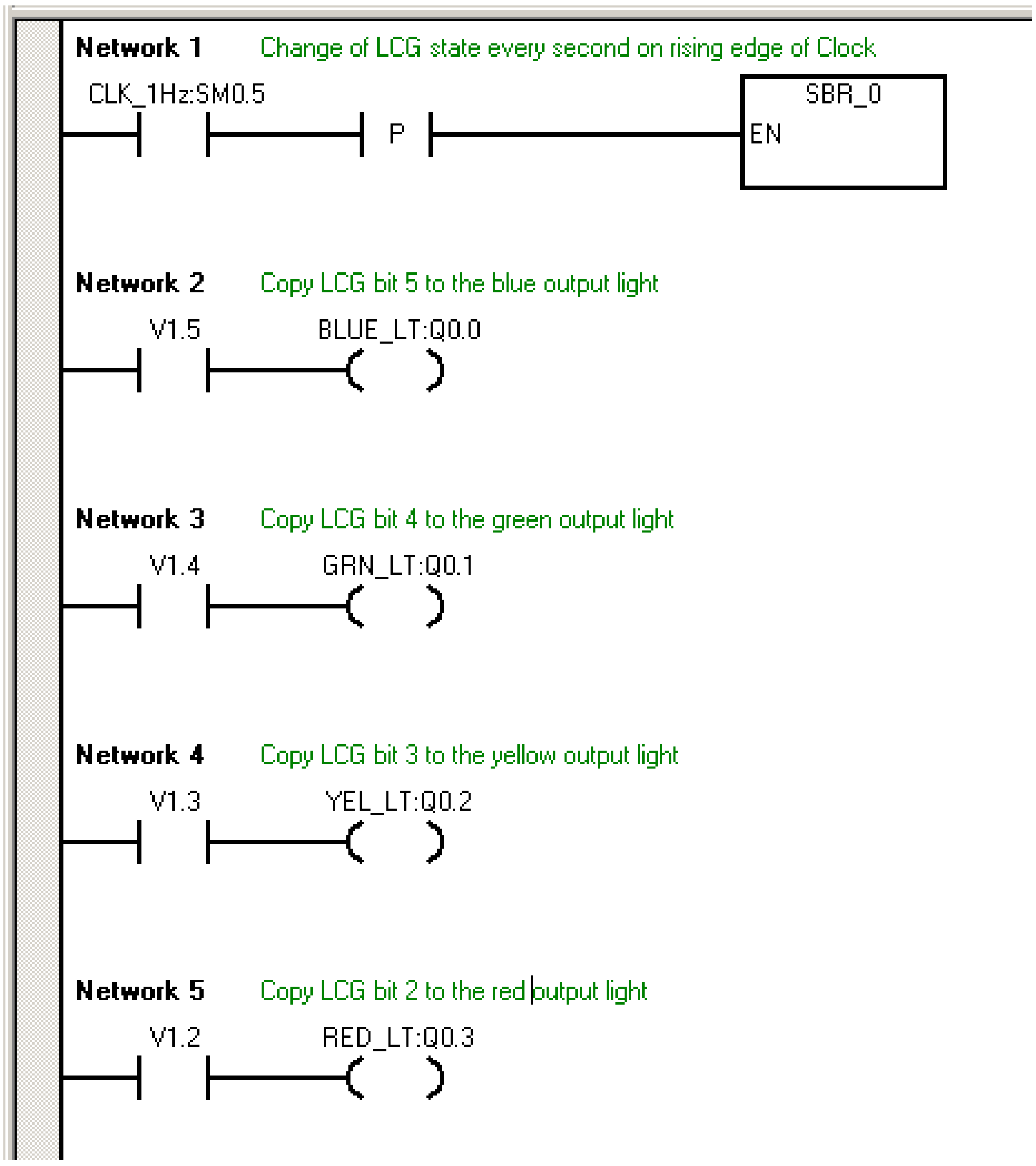
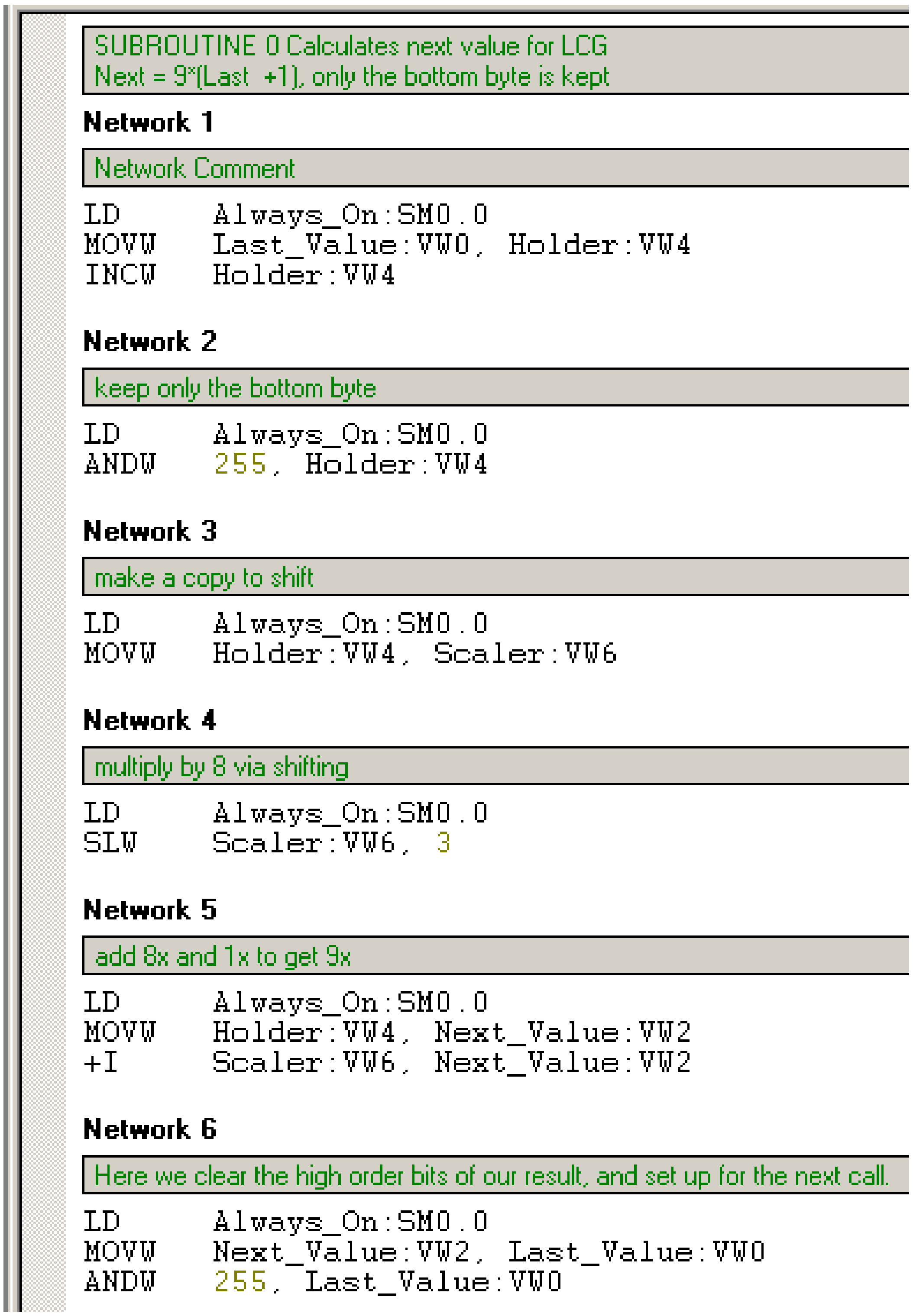

Network 1
Change of LCG state every second on rising edge of Clock
CLK_1Hz:SM0.5
P
SBR_0
EN
Network 2
Copy LCG bit 5 to the blue output light
V1.5
BLUE_LT:Q0.0
Network 3
Copy LCG bit 4 to the green output light
V1.4
GRN_LT:Q0.1
Network 4
Copy LCG bit 3 to the yellow output light
V1.3
YEL_LT:Q0.2
Network 5
Copy LCG bit 2 to the red output light
V1.2
RED_LT:Q0.3

Figure 4: Main Program with One Second Updating



SUBROUTINE 0 Calculates next value for LCG
Next = 9*(Last +1), only the bottom byte is kept

Network 1

Network Comment

```
LD      Always_On:SM0.0
MOVW    Last_Value:VW0, Holder:VW4
INCW    Holder:VW4
```

Network 2

keep only the bottom byte

```
LD      Always_On:SM0.0
ANDW    255, Holder:VW4
```

Network 3

make a copy to shift

```
LD      Always_On:SM0.0
MOVW    Holder:VW4, Scaler:VW6
```

Network 4

multiply by 8 via shifting

```
LD      Always_On:SM0.0
SLW     Scaler:VW6, 3
```

Network 5

add 8x and 1x to get 9x

```
LD      Always_On:SM0.0
MOVW    Holder:VW4, Next_Value:VW2
+I      Scaler:VW6, Next_Value:VW2
```

Network 6

Here we clear the high order bits of our result, and set up for the next call.

```
LD      Always_On:SM0.0
MOVW    Next_Value:VW2, Last_Value:VW0
ANDW    255, Last_Value:VW0
```

Figure 5: Shift and Sum LCG Implementation

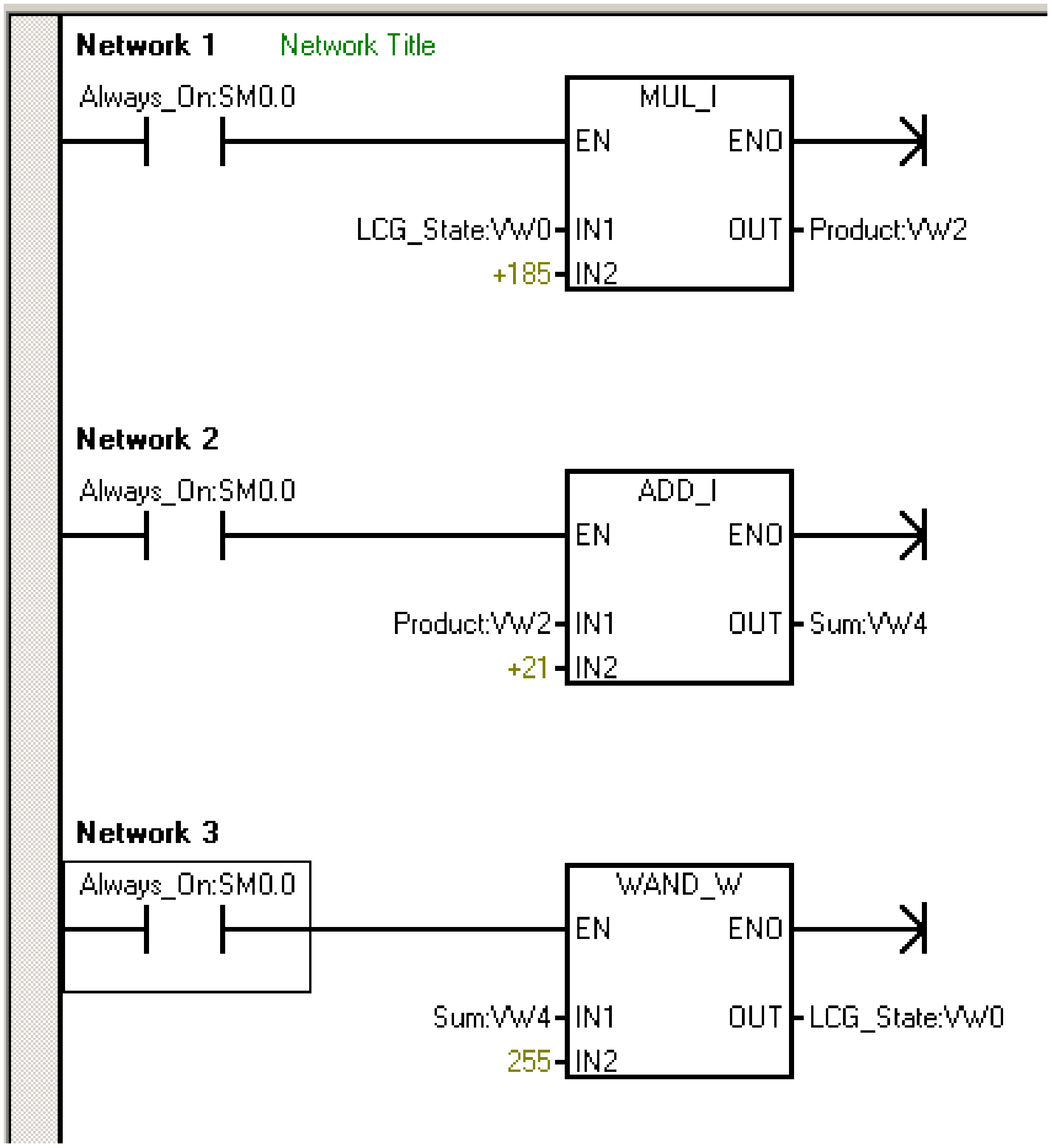


Figure 6: $185x_i + 21$ Implementation Using Ladder Logic

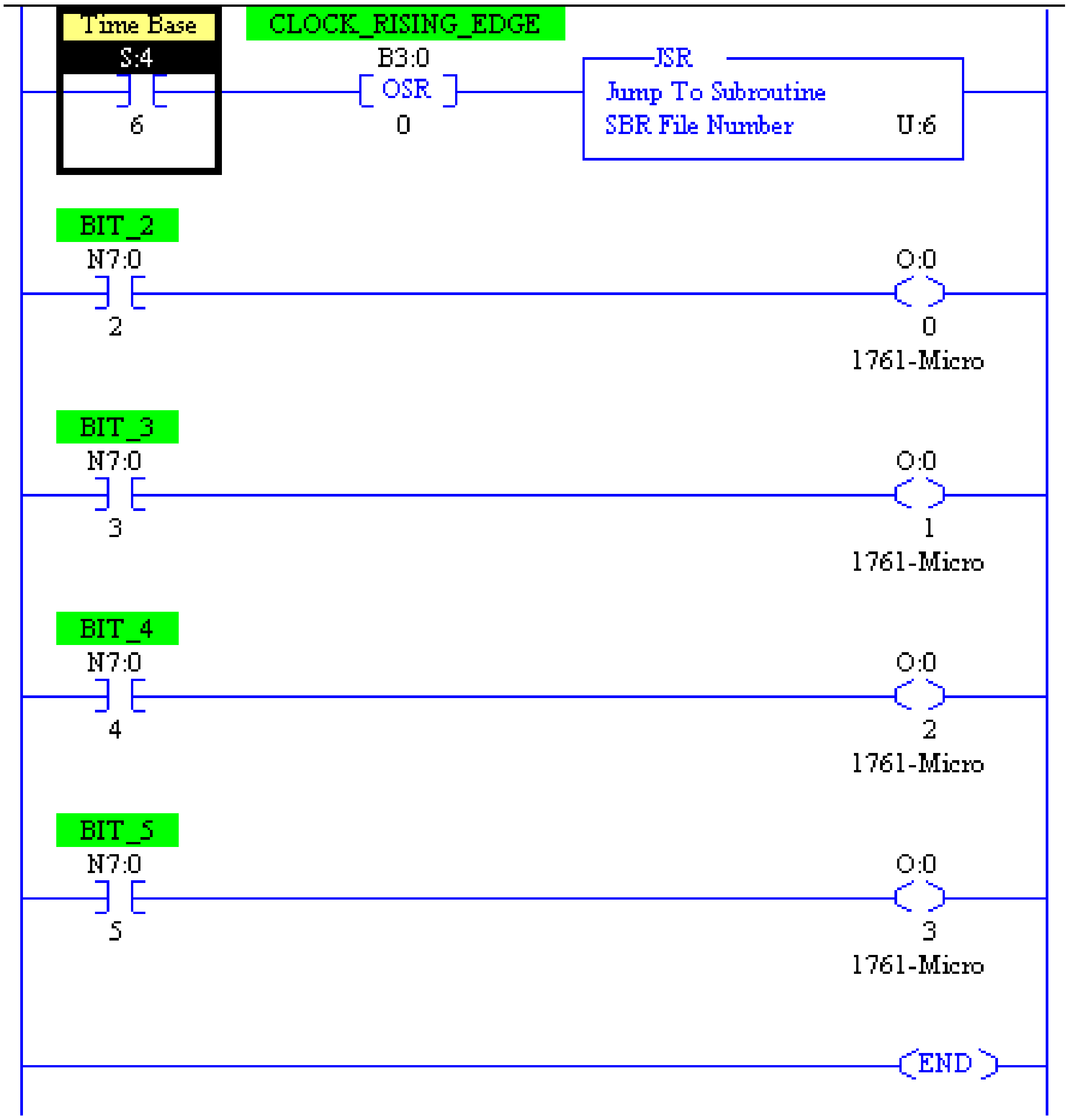
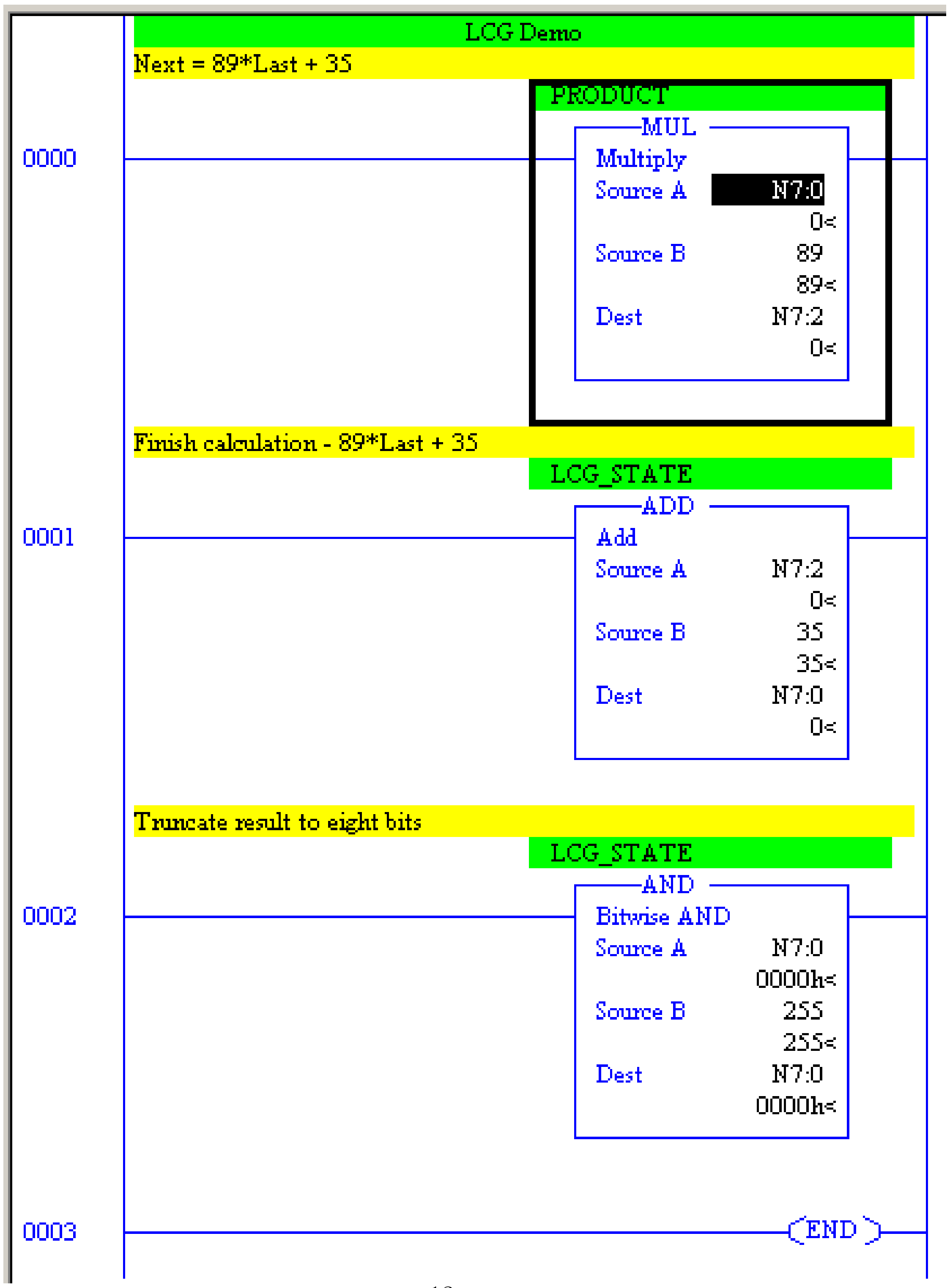


Figure 7: Main Program with About One Second Updating



LCG Demo

Next = 89*Last + 35

```
0000                                   PRODUCT
                                       -MUL
                                       Multiply
                                       Source A        N7:0
                                                         0<
                                       Source B          89
                                                         89<
                                       Dest            N7:2
                                                         0<
```

Finish calculation - 89*Last + 35

```
0001                                   LCG_STATE
                                       -ADD
                                       Add
                                       Source A        N7:2
                                                         0<
                                       Source B          35
                                                         35<
                                       Dest            N7:0
                                                         0<
```

Truncate result to eight bits

```
0002                                   LCG_STATE
                                       -AND
                                       Bitwise AND
                                       Source A        N7:0
                                                     0000h<
                                       Source B         255
                                                        255<
                                       Dest            N7:0
                                                     0000h<

0003 --------------------------------------------(END)---
```

Figure 8: $89x_i + 35$ Implementation Using Ladder Logic