Efficient MLS pseudorandom generator for ARM cpu

Ivan Mellen, Embedded Signals, Apr 2010

A Maximum Length Sequence (MLS) is a type of pseudorandom binary sequence. Its generation is based on maximal Linear Feedback Shift Registers (LFSR).

The usual way of generating MLS is to calculate each output bit individually, then pack output bits into output word with desired width. Since new bit calculation requires at least 3 cycles, 32 bit random number requires at least 96 cycles. Program below can do it in 4 cycles.

Pseudorandom sequence repeats after $2^N - 1$ bits, where $N$ is length of the shift register. Packing bit sequence into words with width $W$ generates unique sequence of $D = 2^N - 1$ words (actually only first $M = (2^N - 1)/W$ words is truly independent, rest of the sequence is shifted version of the first $M$ words).

Five MLS sequence generators {a to e} most suitable for the 32 bit ARM architecture were implemented as described below.

Speed was tested with Cortex M3 based microcontroller. The Cycle column describes time required to calculate complete word, not only single bit of output.

<table>
<thead>
<tr>
<th>Generator</th>
<th>Bits out (word width)</th>
<th>Taps</th>
<th>Cycles</th>
<th>Periode length</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1-28 b</td>
<td>[31 28]</td>
<td>3</td>
<td>$2^{31} - 1$</td>
</tr>
<tr>
<td>b</td>
<td>1-25 b</td>
<td>[32 30 26 25]</td>
<td>5</td>
<td>$2^{32} - 1$</td>
</tr>
<tr>
<td>c</td>
<td>32 b</td>
<td>[63 62]</td>
<td>4</td>
<td>$2^{63} - 1$</td>
</tr>
<tr>
<td>d</td>
<td>32 b</td>
<td>[64 63 61 60]</td>
<td>6</td>
<td>$2^{64} - 1$</td>
</tr>
<tr>
<td>e</td>
<td>64 b</td>
<td>[64 63 61 60]</td>
<td>10</td>
<td>$2^{64} - 1$</td>
</tr>
</tbody>
</table>

Just to put length of discussed MLS into perspective, this is the time after which output words starts to repeat under assumption that pseudorandom words are generated at 10MHz rate (10 million words per second):

<table>
<thead>
<tr>
<th>Sequence length</th>
<th>Periode duration (@10E6 words per second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{31} - 1$</td>
<td>214.75 seconds = 3.58 min</td>
</tr>
<tr>
<td>$2^{32} - 1$</td>
<td>429.49 seconds = 7.15 min</td>
</tr>
<tr>
<td>$2^{63} - 1$</td>
<td>9.2E11 seconds = 29227.1 years</td>
</tr>
<tr>
<td>$2^{64} - 1$</td>
<td>18.4E11 seconds = 58454.2 years</td>
</tr>
</tbody>
</table>
Output bit packing example for generator a \([31 \, 28]\):

\( W \) output (only \( W \) lower bits shown), \( W \) set to multiples of 4 for easier reading

4 bits \( 0 \, 0 \, 0 \, f \, 0 \, 7 \, 2 \, 8 \, 0 \, e \, e \, 7 \, c \, d \, 0 \, 0 \, f \, 3 \, b \, 3 \, 4 \, a \, 0 \, e \, d \, c \, 5 \, ... \)

8 bits \( 00 \, 0f \, 07 \, 28 \, 00 \, ee \, 7c \, d0 \, 0f \, 3b \, 34 \, a0 \, ed \, c5 \, ... \)

16 bits \( 000f \, 0728 \, 00ee \, 7cd0 \, 0f3b \, 34a0 \, edc5 \, ... \)

24 bits \( 000f07 \, 2800ee \, 7cd00f \, 3b34a0 \, edc523 \, ... \)

28 bits \( 000f072 \, 800ee7c \, d00f3b3 \, 4a0edc5 \, ... \)

Text bellow contains C code together with corresponding ARM instructions.

**Note:** variables \( \text{rnd} \) (31,32 bit version) resp. \( \text{rndL, rndH} \) (63,64 bit version) contain generator state, so they must be preserved during generator lifetime. Also, these variable has to be initialized with non zero initial seed.

**C code section:**

```c
unsigned int \text{rnd};        // state of 31 and 32 bit generator
unsigned int \text{rndH,rndL};  // state of 63 and 64 bit generator
unsigned int \text{tmp,tmp2};   // temporary variables
const \text{W}=24; // \( W \) bits generated in 1 iteration , generators a,b
    // \( W=1 \) to 28 for 31 bit MLS (a)
    // \( W=1 \) to 25 for 32 bit MLS (b)

//a) 1-28 bit out; taps \([31,28]\); 3 cycles; length= \( 2^{31}-1 \)

\text{tmp}= \text{rnd} ^ (\text{rnd} \ll 3);          // eor  \text{tmp, rnd, rnd,lsl #3}
\text{rnd}= \text{rnd} \ll \text{W};                 // lsrs \text{rnd, rnd, #W}
\text{rnd}= \text{rnd} | (\text{tmp} >> (31-\text{W})) // orr  \text{rnd, rnd, tmp, lsr #31-W}

output: \( W \) least significant bits of \text{rnd}

//b) 1-25 bit out; taps \([32 \, 30 \, 26 \, 25]\); 5 cycles; length= \( 2^{32}-1 \)

\text{tmp}=\text{rnd} ^ (\text{rnd} \ll 2);          // eor  \text{tmp, rnd, rnd,lsl #2}
\text{tmp2}=\text{rnd} ^ (\text{rnd} \ll 1);       // eor  \text{tmp2, rnd, rnd,lsl #1}
\text{tmp}=\text{tmp}^* (\text{tmp2} \ll 6);     // eor  \text{tmp, tmp, tmp2,lsl #6}
\text{rnd}=\text{rnd} \ll \text{W};                  // lsrs \text{rnd, rnd, #W}
\text{rnd}= \text{rnd} | (\text{tmp} >> 32-\text{W});  // orr  \text{rnd, rnd, tmp, lsr #32-W}

output: \( W \) least significant bits of \text{rnd}

//c) 32 bit out; taps \([63 \, 62]\); 4 cycles; length= \( 2^{63}-1 \)

\text{tmp}= \text{rndH} ^ (\text{rndH} \ll 1);       // eor  \text{tmp, rndH, rndH,lsl #1}
\text{rndH}= \text{rndL} | (\text{tmp} >> 31);      // orr  \text{rndH, rndL, tmp, lsr #31}
\text{rndL}= \text{tmp} ^ (\text{rndL} \ll 31);     // eor  \text{rndL, tmp, rndL,lsl #31}
\text{rndL}= \text{rndL} \ll 1;                    // lsls \text{rndL, rndL, #1}

output:32 bits in \text{rndH}
```
//d) 32 bits out; taps [64 63 61 60]; 6 cycles; length= 2^64-1

    rndH = rndH ^ (rndH <<1);  // eor  rndH, rndH, rndH, lsl #1
    rndH = rndH ^ (rndH <<3);  // eor  rndH, rndH, rndH, lsl #3
    tmp =  rndL ^ (rndL >>1);   // eor  tmp,  rndL, rndL, lsr #1
    tmp =  rndH ^ (tmp >>28);   // eor  tmp,  rndH, tmp,  lsr #28
    rndH = rndL;                // mov  rndH, rndL
    rndL = tmp ^ (rndL >>31);   // eor  rndL, tmp,  rndL, lsr #28

output:32 bits in rndL (or rndH)

//e) 64 bits out; taps [64 63 61 60]; 10 cycles; length= 2^64-1

    rndH = rndH ^ (rndH <<1);  // eor  rndH, rndH, rndH, lsl #1
    rndH = rndH ^ (rndH <<3);  // eor  rndH, rndH, rndH, lsl #3
    tmp =  rndL ^ (rndL >>1);   // eor  tmp,  rndL, rndL, lsr #1
    tmp =  rndH ^ (tmp >>28);   // eor  tmp,  rndH, tmp,  lsr #28
    rndH = tmp ^ (rndL >>31);   // eor  rndL, tmp,  rndH, lsr #28
    rndL = rndL ^ (rndL <<1);   // eor  rndL, rndL, rndL, lsl #1
    rndL = rndL ^ (rndL <<3);   // eor  rndL, rndL, rndL, lsl #3
    tmp =  rndL ^ (tmp >>28);   // eor  tmp,  rndL, tmp,  lsr #28
    tmp =  rndH ^ (tmp >>28);   // eor  tmp,  rndH, tmp,  lsr #28
    rndL = tmp ^ (rndH >>31);   // eor  rndH, tmp,  rndH, lsr #28

output:64 bits [rndH, rndL]

If you have any questions contact address is   imellen@embeddedsignals.com