Application Note: AN01007

Random numbers on the XS1-L1

The XMOS XS1-L8 multicore microcontroller has provisions that facilitate the construction of random numbers. This application note gives a brief introduction to those methods.

1 Overview

The best method to generate pseudo-random numbers is to use a standard library function such as `random(3)` and `srandom(3)`. However, if this is too costly in terms of CPU or memory overhead, the CRC primitive can be used instead. When used with a primitive polynomial, a pseudo-random sequence can be generated by repeatedly executing:

```
crc32(seed, -1, polynomial);
```

Repeatedly executing the CRC using the polynomial 0xEB31D82E generates a sequence:

```
0x842FF083
0x65EB6512
0x93666947
0x98AB23FE
0x2D516F58
0x6BF973E7
0x664F23F0
...
```

Since crc32 is a single machine instruction that executes in a single core cycle, it adds only minimal overhead.

The CRC instruction generates a pseudo-random sequence. The length of the pseudo-random sequence depends on the polynomial chosen. A polynomial of degree p limits the sequence to at most 2^p. To get close to that sequence, one should choose a *primitive polynomial*[1]. Note that polynomials with only few terms (e.g. 0x80001001) have a tendency to create runs of one and zero bits; more information can be found in Jain[2].

[1] http://en.wikipedia.org/wiki/Cyclic_redundancy_check

[2] Raj Jain. The Art of Computer Systems Performance Analysis. John Wiley & Sons Inc, New York, 1991.

2 Real random numbers

If real random numbers are required, there are a number sources of randomness that can be exploited in the XS1-L8: the timer and four ring oscillators. If an external source of events is available (for example, a USB PHY or an ethernet PHY) then any signals from this can be used too.

2.1 Ring oscillators

The four ring oscillators run independent from each other and are independent from the 100 MHz clock. Measuring these timers over, for example, a 50μs period will produce a value between 20000 and 25000, depending on temperature, voltage, and silicon. Between subsequent readings over a fixed time period there will be an uncertainty due to short term variations in voltage and temperature. This is a good source of random data, providing around one bit of random data over a 50μs period on an otherwise quiescent xCORE. On an active tile (with multiple running cores), there will be more random noise, and random data will be available in a shorter time period.

The six statements that control and read the ring oscillators are:

```
setps(0x060b, 0xF); // Enable ring oscillators
setps(0x060b, 0x0); // Disable ring oscillators
r0 = getps(0x070b); // read out oscillator 0
r1 = getps(0x080b); // read out oscillator 1
r2 = getps(0x090b); // read out oscillator 2
r3 = getps(0x0a0b); // read out oscillator 3
```

The `getps` and `setps` functions are defined in the `xs1.h` include file.

An example code sequence that uses the above functions and the 100 MHz timer to generate a four-bit random number is given below. The ring oscillator values are read twice, and the last bit of the difference of each ring oscillator is used to generate a total of four random bits in the last line:

```
timer t;
unsigned time, r;
unsigned short r0, r1, r2, r3;
unsigned short r0a, r1a, r2a, r3a;
r0a = getps(0x070b);
r1a = getps(0x080b);
r2a = getps(0x090b);
r3a = getps(0x0a0b);
setps(0x060b, 0xF); // enable RO
t :> time;
t when timerafter(time+5000) :> int _;
setps(0x060b, 0);   // disable RO
r0 = getps(0x070b);
r1 = getps(0x080b);
r2 = getps(0x090b);
r3 = getps(0x0a0b);
r = ((r0-r0a)&0x1)<<3|((r1-r1a)&0x1)<<2|
   ((r2-r2a)&0x1)<<1|((r3-r3a)&0x1);
```

There is no requirement to wait for precisely 50μs—computations can be performed during this time.

Measuring the ring oscillator against the timer is sufficient for generating unique identifiers or seeding random timeouts. For cryptographic purposes, you are advised to characterize the randomness on the test platform under the desired conditions. Small on-chip variations in voltage and temperature will guarantee a random bit to be generated if measured over long enough a period.

2.2 External events

External events provide a useful source of randomness, as measured against either the 100 MHz timer or the ring oscillator. For example, absolute time at which a USB enumeration occurs is not completely predictable since the USB SOF clock and the 100 MHz timer are asynchronous. If a device is USB powered, then the 100 MHz timer will have a similar value on every enumeration, but the last few bits of the timer will be random.

Similarly, the time between SOF packets, USB transactions, or Ethernet packets can be measured using either the 100 MHz timer or the ring oscillator. Given that the ring oscillators run at a higher frequency, and that they are not locked to a crystal, they are likely to provide more random data than the 100 MHz timer.

CAUTION: The 100 MHz timer is locked to the instruction stream, so sampling the 100 MHz timer in the boot sequence will provide a non-random value. The value of the 100 MHz timer only provides randomness when measured against an external signal.

2.3 Measuring over long time intervals

When measuring the difference in timers over longer time intervals, the number of random bits only increases slowly, and hence random bits are generated more efficiently when sampling the timer *X* times at 50μs intervals, then to sample it once after a 50Xμs interval. Sampling *X* times at 50μs intervals will give you approximately *X* random bits per ring oscillator. Sampling once after a 50Xμs interval will give you approximately $\sqrt{X}$ random bits per ring oscillator: sampling over a long time interval effectively uses random bits to create a binomial distribution which has little variance.

3 Collating multiple random bits

The random bits can be combined by using bit shift operations, by using one of the built-in random number primitives such as srandom(3C), or by using the previously mentioned CRC instruction. As an example, if you want to construct a random number while enumerating a USB device, you can use the following two functions:

```
#include <xs1.h>

unsigned int seed;

/* Call init() to initialize, eg at start of enumeration */
init() {
   timer t;
   int time;
   t :> seed;
   setps(0x060b, 0xF); // Switch on all ring oscillators
}

/* Called on every SOF and every packet */
collectRandom() {
   unsigned short r0, r1, r2, r3;
   setps(0x060b, 0);   // Switch off RO
   r0 = getps(0x070b);
   r1 = getps(0x080b);
   r2 = getps(0x090b);
   r3 = getps(0x0a0b);
   setps(0x060b, 0xF); // Switch RO back on
   crc32(seed, r0, 0xEB31D82E);
   crc32(seed, r1, 0xEB31D82E);
   crc32(seed, r2, 0xEB31D82E);
   crc32(seed, r3, 0xEB31D82E);
}
```

In this example, the polynomial used for computing the CRC of an ethernet packet, is used as a method of cheaply collating the random data, hence the code will collect random data into a 32-bit word. Executed over a long enough period of time, it will collate 32 bits of random data.

At any time, the seed can be used to start a pseudo-random sequence by using the CRC instruction:

```
crc32(seed, -1, 0xEB31D82E);
```

This just generates the pseudo-random sequence, and will not merge in additional true random data.

4 Conclusions

This note describes low-overhead methods for generating random and pseudo-random numbers. The standard random functions `srandom()` and `random()` can be used to generate pseudo-random numbers with well-known and tested properties. If the footprint of those functions is too large, then the `crc32()` primitive can be used as a low-overhead alternative.

Real random can be created by either measuring the on-chip ring oscillators against the built-in timer, or by measuring the timings of external events against either the ring oscillators or the built-in timer. The pseudo-random generators above can be used to collate random bits obtained from an oscillator into a long random state.