Lecture 6: A Random Number Generator in Verilog

A Design Lecture

Patrick Schaumont
What is a random number generator?

Random Number Generator

11, 86, 82, 52, 60, 46, 64, 10, 98, 2, ...
What do I do with randomness?

- **Play games!**
  - Have the monsters appear in different rooms every time

- **Do statistical simulations**
  - Simulate customers in a shopping center (find the best spot for a new Chuck E Cheese)

- **Run security protocols**
  - Make protocol resistant against replay

- **Encrypt documents**
  - Use random numbers as key stream
Encrypt Documents

Random Number Generator

'one-time pad'

XOR

stream of bytes
plaintext

encrypted stream of bytes
cryptext

XOR

decrypted stream of bytes
plaintext
Random numbers by physical methods

- Use dice, coin flips, roulette
- Use thermal noise (diodes and resistors)
- Use clock jitter (use ring oscillators)
- Use radioactive decay
- Use Lava Lamps
  - Patented!

United States Patent

<table>
<thead>
<tr>
<th>Patent Number:</th>
<th>Date of Patent:</th>
</tr>
</thead>
</table>

METHOD FOR SEEDING A PSEUDO-RANDOM NUMBER GENERATOR WITH A CRYPTOGRAPHIC BASE OF A DIGITIZATION OF A CHAOTIC SYSTEM

Inventors: Landon Curt Noll; Robert G. Mendel, both of Sunnyvale; Satyajeet Sisodiya, Mountain View, all of Calif.


Appl. No.: 592,891
Filed: Jan. 29, 1996

Int. Cl.: H04L 9/22
U.S. Cl.: 380/28; 380/46; 364/717
Field of Search: 380/14, 28, 46; 395/421, 06; 364/717

References Cited

A method for generating a pseudo-random numbers. Initially, the state of a chaotic system is digitized to form a binary string. This binary string is then hashed to produce a second binary string. It is this second binary string which is used to seed a pseudo-random number generator. The output from the pseudo-random number generator may be used in forming a password or cryptographic key for use in a security system.
Random numbers by computational methods

- Not truly random, but *pseudo* random
  - meaning, after some time the same sequence returns

- Linear Congruential Generator

\[ x(n+1) = [(\ a \cdot x(b) + b) \mod m] \]

Eg. \( a = 15, b = 5, m = 7 \)

\[
\begin{align*}
X(0) &= 1 \\
X(1) &= (15 + 5) \mod 7 = 6 \\
x(2) &= (15 \cdot 6 + 5) \mod 7 = 4 \\
x(3) &= 2 \\
x(4) &= 0 \\
x(5) &= 5 \\
x(6) &= 3 \\
x(7) &= 1 \\
x(8) &= \ldots
\end{align*}
\]

\( a, b, m \) must be chosen carefully!

*for a maximum length sequence*
A quick way to generate random numbers

- Verilog has a buildin random number generator

```verilog
module random(q);
  output [0:31] q;
  reg [0:31] q;

  initial
    r_seed = 2;

  always
    #10 q = $random(r_seed);
endmodule
```

- Nice, but only for testbenches …

- Instead, we want an hardware implementation
Linear Feedback Shift Register

- Pseudo Random Numbers in Digital Hardware

Diagram:
- Shift register
- Feedback network
Linear Feedback Shift Register

- All zeroes
  - not very useful ...

\[
0 \rightarrow 0 \rightarrow 0 \rightarrow 0
\]
Linear Feedback Shift Register

- Non-zero state is more interesting
Linear Feedback Shift Register

- Non-zero state is more interesting

```
1 0 0 0
```
Linear Feedback Shift Register

- Non-zero state is more interesting
Linear Feedback Shift Register

- Non-zero state is more interesting

\[
\begin{array}{c}
0 \\
0 \\
1 \\
0
\end{array}
\]

1001
Linear Feedback Shift Register

- Non-zero state is more interesting

```
1 0 0 1
```

Patrick Schaumont

Spring 2008
Linear Feedback Shift Register

- Non-zero state is more interesting

[Diagram of a Linear Feedback Shift Register]

100110
Linear Feedback Shift Register

- Non-zero state is more interesting

1001101

etc ...
Linear Feedback Shift Register

- This is actually a finite state machine

State Encoding

1 0 0 1
Linear Feedback Shift Register

- This is actually a finite state machine

How many states will you see?
Linear Feedback Shift Register

- 15 states

Diagram of a 15-state Linear Feedback Shift Register with binary states progression from 0001 to 0011.
Linear Feedback Shift Register

We can specify an LFSR by means of the characteristic polynomial (also called feedback polynomial)

\[ P(x) = x^4 + x^3 + 1 \]

There exists elaborate finite-field math to analyze the properties of an LFSR - outside of the scope of this class.
Linear Feedback Shift Register

- So, knowing the polynomial you can also draw the LFSR.

\[ P(x) = x^8 + x^6 + x^5 + x^4 + 1 \]

How many taps?
How many 2-input XOR?
Linear Feedback Shift Register

- So, knowing the polynomial you can also draw the LFSR

\[ P(x) = x^8 + x^6 + x^5 + x^4 + 1 \]

How many taps? 8
How many 2-input XOR?
So, knowing the polynomial you can also draw the LFSR

\[
P(x) = x^8 + x^6 + x^5 + x^4 + 1
\]

How many taps? 8
How many 2-input XOR? 3
Linear Feedback Shift Register

- Certain polynomials generate very long state sequences. These are called maximal-length LFSR.

\[ P(X) = x^{153} + x^{152} + 1 \]

is a maximum-length feedback polynomial

State machine with \( 2^{153} -1 \) states ..
Fibonacci and Galois LFSR

- This format is called a Fibonacci LFSR

- Can be converted to an equivalent Galois LFSR
Each Fibonacci LFSR can transform into Galois LFSR:
- Reverse numbering of taps
- Make XOR inputs XOR outputs and vice versa

Example: starting with this Fibonacci LFSR
Fibonacci and Galois LFSR

- Disconnect XOR inputs
- Reverse tap numbering (not the direction of shifting!)
Fibonacci and Galois LFSR

- Turn XOR inputs into XOR outputs and vice versa
Which one is better for digital hardware?

- **Fibonacci**

- **Galois**
Which one is better for digital hardware?

- Fibonacci

- Galois computes all taps in parallel
Which one is better for software?

- **Fibonacci**

- **Galois**
Which one is better for software?

- **Fibonacci**

```
/boxshadowdwn
Fibonacci
1 2 3 4 5 6 7 8
```

- **Galois**

```
char_v = (char_v >> 1) ^ -(signed char) (char_v & 1) & 0xe
```

```
/boxshadowdwn
Galois
char_v = (char_v >> 1) ^ -(signed char) (char_v & 1) & 0xe
8 7 6 5 4 3 2 1
```
Let's write an LFSR in Verilog

xor(out, in1, in2)

You need to build this one (structural, behavioral)
module flipflop(q, clk, rst, d);

  input clk;
  input rst;
  input d;
  output q;
  reg q;

always @(posedge clk or posedge rst)
  begin
    if (rst)
      q = 0;
    else
      q = d;
  end
endmodule
A Flip flop

- Setup Time:
  - Time D has to be stable before a clock edge

- Hold Time:
  - Time D has to be stable after clock edge

- Propagation Delay:
  - Delay from clock edge to Q
  - Delay from reset to Q

How to specify propagation delay?
module flipflop(q, clk, rst, d);
  input clk;
  input rst;
  input d;
  output q;
  reg q;

  always @(posedge clk or posedge rst)
  begin
    if (rst)
      q = 0;
    else
      q = d;
  end
endmodule
A Flip flop

```verilog
module flipflop(q, clk, rst, d);
  input clk;
  input rst;
  input d;
  output q;
  reg q;

  always @(posedge clk or posedge rst)
    begin
      if (rst)
        #2 q = 0;
      else
        q = #3 d;
    end

  specify
    $setup(d, clk, 2);
    $hold(clk, d, 0);
  endspecify
endmodule
```

Test setup, hold

See Chapter 10
Palnitkar
Let's turn the LFSR into a module

How to program this?

output (1 bit)
Let's turn the LFSR into a module

- seed (4 bit)
- load (1 bit)
- output (1 bit)
module mux(q, control, a, b);
  output q;
  reg q;
  input control, a, b;
  wire notcontrol;

  always @(control or notcontrol or a or b)
    q = (control & a) | (notcontrol & b);

  not (notcontrol, control);
endmodule;
LFSR, structural

module lfsr(q, clk, rst, seed, load);
  ...
  wire [3:0] state_out;
  wire [3:0] state_in;

  flipflop F[3:0] (state_out, clk, rst, state_in);
endmodule
module lfsr(q, clk, rst, seed, load);

wire [3:0] state_out;
wire [3:0] state_in;
wire nextbit;

xor G1(nextbit, state_out[2], state_out[3]);

assign q = nextbit;

endmodule
module lfsr(q, clk, rst, seed, load);

wire [3:0] state_out;
wire [3:0] state_in;
wire nextbit;

mux M1[3:0] (state_in, load, seed, {state_out[2], state_out[1], state_out[0], nextbit});

assign q = nextbit;
endmodule
LFSR module - complete

module lfsr(q, clk, rst, seed, load);
    output q;
    input [3:0] seed;
    input load;
    input rst;

    wire [3:0] state_out;
    wire [3:0] state_in;

    flipflop F[3:0] (state_out, clk, rst, state_in);
    mux M1[3:0] (state_in, load, seed, {state_out[2],
        state_out[1],
        state_out[0],
        nextbit});

    xor G1(nextbit, state_out[2], state_out[3]);
    assign q = nextbit;

endmodule
module lfsrtest;
    reg clk;
    reg rst;
    reg [3:0] seed;
    reg load;
    wire q;
    lfsr L(q, clk, rst,
            seed, load);

    // initialization
    // apply reset pulse
initial
    begin
        clk = 0;
        load = 0;
        seed = 0;
        rst = 0;
        #10 rst = 1;
        #10 rst = 0;
    end
endmodule

// drive clock
always
    #50 clk = !clk;

// program lfsr
initial begin
    #100 seed = 4'b0001;
    load = 1;
    #100 load = 0;
end
endmodule
Simulation ..

ECE 4514 Digital Design II
Lecture 6: A Random Number Generator in Verilog
Place and Route ..
Can a random number generator have flaws?
Can a random number generator have flaws?

- **Problem #1: it can have bias**
  - Meaning: a certain number occurs more often than others
  - Expressed in the *entropy rate* of the generator
  - Entropy = true information rate (in bit/sec), can be lower than the actual bitrate of the random number generator
  - Example: Assume an RNG that produces three events A, B, C encoded with two bits

<table>
<thead>
<tr>
<th>symbol</th>
<th>probability</th>
<th>bitpattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>P(A) = 1/2</td>
<td>00</td>
</tr>
<tr>
<td>B</td>
<td>P(B) = 1/4</td>
<td>01</td>
</tr>
<tr>
<td>C</td>
<td>P(C) = 1/4</td>
<td>10</td>
</tr>
</tbody>
</table>

E.g. ABACAABC... is encoded as 0001001000000110...

So this bitstream has much more '0' than '1'. It has a bias.
Can a random number generator have flaws?

- **Problem #1:** it can have bias
  - Meaning: a certain number occurs more often then others
  - Expressed in the *entropy rate* of the generator
  - Entropy = true information rate (in bit/sec), can be lower then the actual bitrate of the random number generator
  - Example: Assume an RNG that produces three events A, B, C encoded with two bits

<table>
<thead>
<tr>
<th>symbol</th>
<th>probability</th>
<th>bitpattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>P(A) = 1/2</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>P(B) = 1/4</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>P(C) = 1/4</td>
<td>11</td>
</tr>
</tbody>
</table>

RNG

E.g. ABACAABC... is encoded as 010011001011...

In this bitstream, the number of '1' and '0' are balanced.
Can a random number generator have flaws?

- Problem #1: it can have bias
  - Do LFSR have a bias?
Can a random number generator have flaws?

- Problem #1: it can have bias
  - Do LFSR have a bias?
  - Yes, they have a small bias because the all-zero state never appears.
  - However, for a very long LFSR, the bias becomes negligible.
Can a random number generator have flaws?

Problem #2: it can be predicted

- Not when truly random physical phenomena
- But, if it is a Pseudo RNG (like an LFSR), it is a deterministic sequence.

Is this really a problem? Yes!

- Don't want to use a predictable RNG for dealing cards, driving a slot machine, ... (at least not if you own the place).
- Don't want to use predictable RNG in security. Predicability = weakness
Can a random number generator have flaws?

- Problem #2: it can be predicted
  - The real issue for PRNG is: can the value of bit N+1 be predicted when someone observes the first N bits.

```plaintext
010111011...
```

The next one will be .. 1 !
LFSR are very predictable..

- Predicting the next output bit is equivalent to knowing the feedback pattern of the LFSR and the states of all LFSR flip-flops.

- Mathematicians (Berlekamp-Massey) found that:
  - Given an N-bit LFSR with unknown feedback pattern, then only 2N bits are needed to predict bit 2N + 1

- So let's say we have an 8-bit LFSR, then we need only 16 bits of the RNG stream before it becomes predictable

- LFSR are unsuited for everything that should be unpredictable
To be unpredictable, the LFSR should be long

- **Solution 1**
  
  Use a maximal-length $P(x) = x^N + \ldots + 1$
  
  with $N \gg (\text{E.g. } 65,536)$
  
  Very expensive to make! 64K flip-flops ..

- **Solution 2: Non-linear Combination Generator**

  ![Diagram of Non-linear Combination Generator]

  Out = $A \oplus B \oplus C$

  *Eg.* out = $AB \oplus BC \oplus C$
43-bit LFSR defined by
- \( P(X) = X^{43} + X^{41} + X^{20} + X + 1 \) => 3XOR, 43 taps
- Maximal Length: \( 2^{43}-1 \)
- Bias \( \sim 2^{-43} \) (because all-zero pattern cannot appear)

37-bit cellular automata shift register
- Combines previous and next statereg into current state reg
- Similar to 37 intertwined state machines (automata)
- Maximal Length: \( 2^{37}-1 \)
- Bias \( \sim 2^{-37} \)
Sample Implementation

- On opencores you can find an implementation of Tkacik's design - *assigned reading of today*
  - (this design has a few minor differences with the spec written by Tkacik - but OK for our purpose)
Module interface

module rng(clk, reset, loadseed_i, seed_i, number_o);
input clk;
input reset;
input loadseed_i;
input [31:0] seed_i;
output [31:0] number_o;
reg [31:0] number_o;
reg [42:0] LFSR_reg;  // internal state
reg [36:0] CASR_reg;  // internal state
always (.. CASR ..)
always (.. LFSR ..)
always (.. combine outputs ..)
endmodule
LFSR Part

```verilog
reg[42:0] LFSR_varLFSR; // temporary working var
reg outbitLFSR; // temporary working var
always @(posedge clk or negedge reset)
    begin
        if (!reset)
            begin
                ...
            end
        else
            begin
                if (loadseed_i)
                    begin
                        ...
                    end
                else
                    begin
                        ...
                    end
            end
    end
```
LFSR Part

```verilog
reg[42:0] LFSR_varLFSR; // temporary working var
reg outbitLFSR;       // temporary working var
always @(posedge clk or negedge reset)
  begin
    if (!reset )
      begin
        LFSR_reg  = (1);
      end
    else
      begin
        if (loadseed_i )
          begin
            LFSR_varLFSR [42:32]=0;
            LFSR_varLFSR [31:0]=seed_i ;
            LFSR_reg  = (LFSR_varLFSR );
          end
        else
          begin
            ...          
          end
      end
  end
```

assemble bits

```
LFSR_varLFSR

LFSR_reg
```

ECE 4514 Digital Design
Lecture 6: A Random Number Generator in Verilog
Patrick Schaumont
Spring 2008
LFSR Part

```verilog
reg[42:0] LFSR_varLFSR; // temporary working var
reg outbitLFSR; // temporary working var
always @(posedge clk or negedge reset)
  begin
    if (!reset )
    else
      begin
        if (loadseed_i )
        else
          begin
            LFSR_varLFSR = LFSR_reg;
            LFSR_varLFSR [42] = LFSR_varLFSR [41];
            outbitLFSR = LFSR_varLFSR [42];
            LFSR_varLFSR [42] = LFSR_varLFSR [41];
            LFSR_varLFSR [41] = LFSR_varLFSR [40]^outbitLFSR ;
            // some lines skipped ...
            LFSR_varLFSR [0] = LFSR_varLFSR [42];
            LFSR_reg = LFSR_varLFSR;
          end
      end
  end
end
```
CASR Part (similar ...)

//CASR:
reg[36:0] CASR_varCASR,CASR_outCASR;
always @(posedge clk or negedge reset)
begin
  if (!reset )
    begin
      ...
    end
  else
    begin
      if (loadseed_i )
        begin
          ...
        end
      else
        begin
          ...
        end
    end
end
//CASR:
reg[36:0] CASR_varCASR,CASR_outCASR; // temp
always @(posedge clk or negedge reset)
begin
  if (!reset )
    begin
      CASR_reg = 1;
    end
  else
    begin
      if (loadseed_i )
        begin
          CASR_varCASR [36:32]= 0;
          CASR_varCASR [31:0] = seed_i ;
          CASR_reg  = (CASR_varCASR );
        end
      else
        begin
          ...
        end
    end
end
CASR Part (similar ...)

//CASR:
reg[36:0] CASR_varCASR,CASR_outCASR; // temp
always @(posedge clk or negedge reset)
    begin
    if (!reset )
    else
        begin
        if (loadseed_i )
        else
            begin
            CASR_varCASR     = CASR_reg ;
            CASR_outCASR [36]= CASR_varCASR [35]^CASR_varCASR [0];
            CASR_outCASR [35]= CASR_varCASR [34]^CASR_varCASR [36];
            // ... some lines skipped
            CASR_reg         = CASR_outCASR;
            end
        end
    end
end
always @(posedge clk or negedge reset)
begin
if (!reset )
begin
    number_o  = (0);
end
else
begin
    number_o  = (LFSR_reg [31:0]^CASR_reg[31:0]);
end
end
Simulation..
Simulation (looking at the state registers)
Place and Route ..
Random number generators
  - Many useful applications

Linear Feedback Shift Registers: PRNG
  - Fibonacci and Galois
  - Maximal-length LFSR
  - Structural Verilog Model

Flaws of Random Number Generators
  - Bias
  - Predictability

Nonlinear Combination Generator
  - Design by Tkacik
  - Behavioral Verilog Model