Abstract

A Pseudo-Random Code Generator is a device capable of generating a large, repeating stream of pseudo-random bits suitable for mixing with other data streams, as would be done in a spread spectrum communications device. Our project, built under the auspices of an NSF grant issued to Bradley University, is a 24-stage code generator, designed using 2-micron SCNA CMOS technology. The design consists of 24 cascaded delay-XOR stages, clocked off of a single synchronous clock. It was designed using Magic and LEdit, and simulated at logical, gate, and transistor levels using custom written C programs, MATLAB, LogicWorks and Spice.

Mathematical Background

Pseudo Random Numbers or PRN’s are used in a variety of applications in the digital electronics industry. A few applications include computer simulation of random events, data encryption, and spectrum spreading. As the demand for the unique properties of PRN’s increases, the need for PRN generators is likewise increasing. The 24-Stage Pseudo Random Code Generator implemented in this project is suitable for use as a code sequence generator in a communications application. Before the many uses of this integrated circuit can be appreciated, an understanding of the mathematics behind code generation must be achieved.

A code generator consists of a multiple stage shift register, with taps (outputs) taken from various stages are mathematically combined and fed back into the shift register (SR). The traditional approach, as shown in Figure 1, involves simple modulo-2 addition of the taps (implemented as an exclusive-OR, XOR), and the result is wired as the input to the SR.
As long as there is at least one non-zero bit in the shift-register at the start of code generation, a sequence of 1’s and 0’s will be generated. Whether this code repeats or not, or is maximal-length, depends on which taps are used. If the generated code causes the shift-register to take on all possible state combinations for its individual stages before repeating, then the code is considered "maximal."

Each bit that is output from the code generator is called a chip. If a shift register with n stages is used, then the maximum number of chips in a non-repeating binary sequence from the generator is $2^{n-1}$ chips, after which the code repeats.

Maximal length codes have several interesting properties:

- There is one more logic one than logic zeros in a full PRN sequence. In other words, if there are five SR stages in a given code generator, the maximal sequence will contain 31 chips, of which 16 will be ones and 15 will be zeros. This is an important property which will be revisited later.
- The length and number of consecutive ones or zeros in a PRN code is a constant for a given SR length. The relative ordering of these runs may vary with tap selection.
- In a bipolar signalling scheme (-1 is assigned to logic zero, +1 is assigned to logic one), the autocorrelation function of a maximal length code is -1 for all values of shifting except within one chip of zero. Within that narrow interval, the autocorrelation function makes a sharp peak whose height is equal to the length of the code.
- Any tap taken from an SR stage in a maximal generator will reveal the same PRN as the output, only shifted in time. Additionally, if this tap is XORed with the output, the result will once again be the original PRN shifted in time by some different time value.
- If a sample window of a length equal to the number of SR stages is taken from the output of a maximal code generator once every cycle, every possible binary combination (except the all-bits-zero case) will appear once, and only once during the period of the PRN.

The first property noted above is especially useful in communications applications. If bipolar signalling is used, the DC component of a narrow-band data signal is reduced by $1/C$, where C equals the code length. In an AM (Amplitude Modulation) broadcast configuration, the DC component dictates the amount of broadcasting power spent on broadcasting the carrier. When longer code lengths are used to modulate the data signal, the energy present at the carrier frequency is significantly reduced.
Top-Level Design Overview

Our design is a modified implementation of a textbook code-generator design. Our modification, as shown in Figure 2, is straightforward. Rather than feeding back "tapped" data XOR’ed through a tree of XOR gates into a singular input, we have reversed the direction of data flow so that a single output gets fed into multiple tap points between stages of a shift register.

![Figure 2. Our modified Code Generator approach, using the same tap structure as Figure 1.](image)

The drawback of this approach over the textbook approach is that there is one XOR gate for every D Flip-Flop in the shift register, regardless of the number of taps. The chief benefit, however, is that the wiring structure changes minimally as tap count or tap structure changes. Additionally, with our method there is a maximum of one gate delay between when the data is available on the feedback path, versus two or more gate delays in the "XOR Tree" in the textbook method. Given that we are attempting to design a rather generic and universal code generating device, we felt that these benefits mitigated our design decision.

In order to verify our design idea, we worked through two independent proof-of-concept simulations. First, we used a C-program (Appendix A, Listing 1) to implement our modified algorithm. We compared our output for several different tap structures to output generated by the textbook algorithm. Next, we built a LogicWorks simulation of a small, 3-stage code generator (Appendix B, Simulation 1) to further prove the workability of our design.

Lower-level Gate Design

With two separate proofs-of-concept verifying our initial design concept, we decided to move forward with the lower-level gate design. At this point, our project consisted of two primary components: XOR gates, and D Flip-Flops. Since XOR gates were already reduced a single-gate level, we concentrated on the design of our D Flip-Flops.

For the basic design of our D Flip-Flops, we used the textbook approach of building a master-slave flip-flop from NAND gates. The separate master and slave sections are built from a clocked SR Flip-Flop, as shown in Figure 3a. Figure 3b illustrates how these stages are combined into a complete master-slave flip-flop. Included in the resulting master-slave flip-flop, then, are inputs for Clk, /Clk, D, /D, /Set, and /Reset as well as the complementary outputs Q and /Q. So long as D and /D are the input data and inverted input data, the device acts as a D Flip-Flop.
Once we had designed this, we also simulated it in LogicWorks. Because our design did not require a
/Reset input, we omitted it from this simulation and from our final design. (Only the /Set input was kept,
as resetting all of the D Flip-Flops to zero would actually cause the code generator to cease generating
codes. A necessary condition for code generation is the presence of at least one non-zero bit within the
D Flip-Flop stages.) As shown in Appendix B, Simulation 2, our LogicWorks simulation verified the D
Flip-Flop characteristic of our circuit.

**Transistor-level Gate Design**

Now that the gate-level design of our devices was complete, the transistor level design of our chip
began. We considered many different layouts for each transistor-level cell, weighing various factors
such as size, speed, and workability. After many levels of redesign, we eventually constructed the gates
shown below.

**CMOS Inverter gate**

The simplest of all the gates, the inverter, is pictured in Figure 4a. As its Spice simulation results show,
Appendix B, simulations 3a & b, the device is fairly quick, having a switching time of approximately 0.2ns
w/out load. With like-sized load, this delay changes little, expanding out to barely 0.4ns. As one can see, this is a very fast, simple device.

(For sake of comparison with my other gates, I am choosing this device as the definition for 1 CMOS
load, and as the standard unit for CMOS drive capability. Table 1 below tabulates the simulation results
for each of the cells designed. Input loading and output driving are determined by the ratio of the
time-constants of the circuit being evaluated to the time-constants of the simple inverter. A 10k-ohm
resistor in series with the input, and a 1pF capacitor in parallel with the output were used to make these
measurements.)

**CMOS XOR gate**

When implementing more complicated multiple-input CMOS gates, one first must calculate
the inverse and complement-of-inverse forms of the logic function being implemented. Then, the
P-transistors are wired from VDD to output as dictated by the complement-of- inverse function,
and the N-transistors are wired from GND to output as dictated by the inverse function.

With this in mind, the XOR gate was designed next. This gate was designed in two parts due to its complex nature. Before we could even begin considering its transistor-level implementation, we needed to decompose the XOR function into AND and OR gates using a set of sum-of-products expressions as follows:
- $A \text{ xor } B = (A \text{ and } \neg B) \text{ or } (\neg A \text{ and } B)$
- $A \text{ xnor } B = (A \text{ and } B) \text{ or } (\neg A \text{ and } \neg B)$
- $\neg (A \text{ xnor } B) = (A \text{ and } \neg B) \text{ or } (\neg A \text{ and } B)$

The first equation above gives the AND-OR definition for XOR, the second gives its inverse, XNOR, and finally the third gives the complement of the inverse. Note that the first and third equations are equivalent.

Upon inspection of these equations, it becomes apparent that all four minterms (a product of inputs to a logic function, one or more of which may be inverted) are required to generate this function. This implies that $A$, $B$, $\neg A$, and $\neg B$ all need to be present to generate $A$ XOR $B$. For this reason, the XOR was split into the XOR body and the inverter-pair, in Figures 4b, and 4c respectively.

The inverter pair, being built from the previously designed inverter circuit, did not require additional simulation. No changes to its various circuit parameters were made. For the complete XOR gate, we performed a separate set of simulations, as detailed in Appendix B, simulations 4a, b, and c.

**Master-Slave Flip-Flop**

Finally came the design of the master-slave flip-flop. This design was approached in a modular manner, such that the clocked SR Flip-Flop was designed first, and then merely duplicated twice to create the master-slave device. For that reason, only the SR’s design is detailed.

The basic unit of the SR Flip-Flop design, as mentioned previously, was the NAND gate. Using the above procedure, we generate the following functions:

- $D_{\text{ clk}} = \neg (\neg D \text{ and } \text{ clk})$
- $\neg D_{\text{ clk}} = \neg (D \text{ and } \text{ clk})$
- $\neg Q = \neg (Q \text{ and } D_{\text{ clk}})$
- \(Q = \neg(\neg Q \text{ and } \neg D_{\text{clk}} \text{ and } \neg \text{Set})\)
- \(!Q = \neg(Q \text{ and } D_{\text{clk}})\)

This results in the design pictured in Figure 4d. So, by building these individual NAND gates together, I built the SR Flip-Flop shown finally in Figure 4e. (Note: There is a three-input NAND present in the design, which is a simple extension of the two-input form. Its derivation is not given.)

It is worth noting that the two clock inputs are labeled 1 and 2. This notation would imply the two-phase non-overlapping clock structure utilized by Mead & Conway primarily for NMOS designs. For our design, this is actually not the case. 1 and 2 are roughly inverse of each other, but labeled as separate phases since they are distributed through separate distribution networks as is described later.

Once the SRFF was designed, we simulated it at a fairly high (50MHz) clock rate, sending through an alternating data stream. Appendix B, Simulation 6(c) shows these results. Simulations 7 and 8 show the results when the SRFF was connected as a full Master-Slave flip flop w/XOR gate, and with four such stages cascaded, respectively. As is apparent from the simulation results, the gate operation was satisfactory even at speeds of 25MHz and 50MHz. This is with a "perfect" clock, however.

*Driver Circuits*

Our next concern was driving circuits. By far, the simplest driving circuits to implement are merely modified inverters. For our driving circuits, we opted to gang up multiple inverters into driving blocks, while simultaneously extending the W/L ratio of the transistors of the inverter. After several informal design tests, we settled on three separate driver types, Driver2x (DRV_2), Driver4x (DRV_4), and Driver6x (DRV_6).

Driver2x is just the XOR_B stage, with the inputs and outputs connected together to create a composite stage. Driver4x is similarly constructed of two inverters, with its transistors stretched to twice the W/L ratio for both N and P diffusion. Driver 6x is constructed of three inverters, also stretched to the same W/L as Driver4x. The intention is to use these in a tiered arrangement to minimize total gate delay. Appendix B, Simulations 9 and 10 highlight the performance of these devices separately.
### Table 1. Summary of cell parameters

<table>
<thead>
<tr>
<th>Cell Name</th>
<th>Input Load</th>
<th>Output Drive</th>
<th>Gate Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>No load</td>
</tr>
<tr>
<td>Inverter</td>
<td>1.0*</td>
<td>1.0*</td>
<td>0.3ns</td>
</tr>
<tr>
<td>XOR</td>
<td>2.0</td>
<td>0.638</td>
<td>0.9ns</td>
</tr>
<tr>
<td>SRFF D input</td>
<td>1.1</td>
<td>0.205</td>
<td>2.4ns</td>
</tr>
<tr>
<td>SRFF Clk input</td>
<td>2.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSFF**</td>
<td>----</td>
<td>----</td>
<td>4.7ns</td>
</tr>
<tr>
<td>DRV2</td>
<td>2.44</td>
<td>1.85</td>
<td>0.3ns</td>
</tr>
<tr>
<td>DRV4</td>
<td>4.44</td>
<td>2.84</td>
<td>0.3ns</td>
</tr>
<tr>
<td>DRV6</td>
<td>6.27</td>
<td>4.81</td>
<td>0.3ns</td>
</tr>
</tbody>
</table>

*By definition

**Input characteristic is same as SRFF

Driving capability is calculated as the ratio between output time constants after subtracting the unloaded single-inverter gate delay of 0.3ns from each time constant.

### Chip Hierarchy Design

With all of the low level gates designed and simulated in LogicWorks and Spice, we moved on to putting the low-level CMOS gates together into a hierarchial chip design. The Magic VLSI Layout Editor supports a hierarchial design paradigm. What this means, essentially, is that it allows us to build our design up from smaller pieces into larger pieces, and further build still larger sections from these larger pieces, and so on, until the entire chip is complete. To put this into perspective, Figure 5 gives an overview of the chip hierarchy.

### The DX Flip-Flop Stage

The smallest homogeneous unit in our ideal code generator, referring back to our top-level design, is the combination of a D Flip-Flop and XOR gate. So, for the lowest level of our main chip hierarchy, we built what we refer to as the "DX Flip-Flop Stage." This stage consists basically of a D Flip-Flop followed by an XOR gate.

Because the XOR gate is actually composed of two halves, XOR_A and XOR_B, we will actually break this up into three pieces, so that we have the trailing half of the previous XOR, followed by the Master-Slave Flip-Flop (acting as a D Flip-Flop), followed lastly by the front half of the next XOR. This allows us to daisy-chain gates readily, with only half of the first half of an XOR being unutilized at the beginning of the chain. (Recall that the second half of the XOR gate, XOR_B, is actually two inverters,
and that for the Master-Slave Flip-Flop to act as a D Flip-Flop, the data input must be inverted. So, one of the two inverters does get used, and the other just has its input tied to ground to prevent it from oscillating or idling in the active state.) Figure 6a illustrates this stage at the cell level.

The 2DX Stage

The next step in our hierarchy was to place two of the DX stages cross-wise with each other. Our goal in this chip layout is to have as dense a gate structure as possible. To aid this, we opted for an S-shaped physical datapath for our circuit. So, we placed one DX stage facing to the left, and a second facing to the right, with the output of the first connected around the left end to the input of the second. Further, the four set inputs were wired together into one composite input, and a DRV_2 stage was inserted to drive them. Figure 6b illustrates this stage at the cell level.

The 8DX-Bus Stage

With 24-stages slated for our completed design, we decided to break our design into three sections of 8 stages. So, the next logical unit for building up our design was the 8DX stage. This stage is merely four 2DX stages tiled together in a four-cell array, with connections added on the right-hand side.

Next, driver circuits, consisting of two pairs of Driver4x circuits, were added to drive both sets of clock signals for all eight DX stages of this composite stage. An additional Driver2x pair was added to the set input as well. Further, a multiconnector bus was drawn in to facilitate the daisy-chaining of these stages in a linear fashion, to simplify the interconnection of these stages into a 24-stage design. Figure 6c illustrates this configuration at the cell level.

Clocking Structure

It was during the design of this stage that we decided to consider the clocking structure of the circuit. Given that we are planning on driving 24 stages from the same clock input, it became readily apparent that something would need to be done regarding clock distribution.

Each MSFF stage has two clock inputs, each of which looks like approximately 2.2 CMOS loads. So, for each clock line, we would have 52.8 CMOS loads being driven. We opted for a cascaded inverter setup for distributing the clocks. To minimize switching transients along the power supply rails, we decided to distribute the clocks, wired in a tiered fashion. Figure 7 details the finished clocking design.

Our intention in building the clock driving tier was to have the ratio of input loads between consecutive tiers be relatively close to $e$, as it can be shown mathematically that a uniform ratio of $e$ between stages of cascaded inverters minimizes overall delay. Our implementation has actual ratios ranging from 2.7 to 4.4.

We starting building our clocking structure from driven end back towards the clock input. Each 8-stage section represents 17.6 CMOS loads on each of 1 and 2. Our solution was to place 2 Driver4x stages at the top and bottom of each 8DX block, tying two such drivers to each stage. Further, we provided leads for interconnection of 1 and 2 lines between different 8DX stages, to equalize minor variances between clock drivers. This arrangement provided an 5.68:17.6 ratio, or 1:3.09 ratio of input loads for this tier of the clocking structure, and permitted ready compensation for minor device variation.
Next, we considered the intermediate tier of our clocking structure. At this tier, we placed two Driver6x stages for each phase, with both Driver6x stages tied together at the outputs driving all 6 Driver4x inputs within each phase. Again, all connections within the tier were tied together to limit timing variances between sections. This arrangement provided a 9.68:26.64, or 1:2.77, ratio of input loads.

Finally, the topmost tier of the structure was considered. For 2, there was only a connection directly to the clock input pin on the pad frame. For 1, we added a single Driver4x stage, to provide phase inversion with respect to 2. At this stage, we had a 2.84:12.54, or 1:4.4 ratio of driver input loads.

**Overall chip**

With all of these pieces designed and built, we then proceeded to put these together into a complete chip design. First, we daisy chained the three 8DX-Bus stages together to form the 24-Stage body. Then we attached the upper tiers of the clocking structure outside of these stages. Next, we wired the Vdd and GND busses to everything. Finally, we wired all of the inputs and outputs to their corresponding pins on the chip pad frame.

In the completed design, we labelled the D input of the first stage as "T0", and then labelled subsequent XOR inputs as "T1" through "T24." T24 is not a proper tap, per se, as there is only an XOR gate between T24 and the output of the circuit, Q. T24 is useful for "perturbing" the output/feedback data stream for the purpose of aligning codes.

Once all of the inputs and outputs were assigned and wired, we had a few pins left over. So, we decided to place a couple extra gates in our chip design. We placed in our design a standalone XOR gate and a standalone MS Flip-Flop (wired as a D Flip-Flop.)

**Final Simulation of Design**

Now that our design was practically complete, we were faced with the task of simulating the completed design to check for errors. This was not a simple task whatsoever. For this, we used Spice once again, running on a dedicated Linux workstation with 52MB of RAM and 384MB of swapspace.

To test our design, we used three basic test suites. The sheer magnitude of our design limited the scope of the tests that could be performed within our limited time frame. The tests can be summarized as follows:

- **The Walking Bit Test** -- All of the stages are initialized into the "set" state, and a zero is walked through all of them. All of the toggle inputs are tied to GND.

- **The Alternating Bit Test** -- All of the stages are initialized into the "set" state, but are made to toggle between one and zero, as all of the toggle inputs are tied to VDD. Since T0 is also tied to VDD, successive stages take on alternate but steady states.

- **Simple Code Generation** -- A tap structure for a 3-stage code is wired into place and verified. All other tap inputs are wired to their corresponding stage’s outputs so that each of the unused stages goes to zero after one clock. All stages are initially placed into the "set" state.

Each of these tests was performed without the pad frame present. Very limited informal tests were done
while the chip was in the frame as well. Although the partial results received were promising, none of the simulations completed due to lack of computer resources. (The simulations w/ frame demanded too much real RAM.) Appendix B, Simulation 12 shows the results of the above three tests without the frame.

**Testing Procedures for Fabricated Chip**

To verify the operation of the fabricated chip, we recommend the following procedures:

- **Smoke Test** -- Connect the chip to VDD and GND on pins 30 and 10 as appropriate. Decouple the chip with at least a .1µF capacitor. Power it up briefly. If no smoke escapes, it passes this phase.

- **Delayed Zero** -- Leave the chip connected to VDD and GND as before. Tie all toggle inputs to GND. Connect a 10K pullup resistor to "/Set" (Pin 4), and a momentary contact switch between "/Set" and GND. Connect a 0-5v pulse train to "Clk" (Pin 36), set to a relatively low (1 Hz) frequency. Power up the chip, and watch the output (Pin 1). Upon pressing the switch, the output should go high, and then approximately 24 clocks after the switch is released, it should go low.

- **Alternating Bit Test** -- Tie all toggle inputs high. Leave the VDD, GND, Clock, and /Set connections as above. Repeat the Delayed Zero test. This time, the output should alternate between high and low for 24 clocks, and then fix itself in a steady state.

- **Simple Code Generation** -- Tie taps T0 through T20 low. Tie T21 and T22 to the chip output. Tie T23 and T24 low. Leave the VDD, GND, Clock, and /Set connections as above. Repeat the Delayed Zero test. This time, a seven-chip random sequence should appear on the output, repeating indefinitely.

- **Extra XOR Testing** -- Using the same VDD and GND configuration, connect a pair of digital inputs to pins 17 and 18, and watch the output on pin 19. The relationship between these inputs and the output should be that of an XOR gate.

- **Extra D Flip-Flop Testing** -- Using the same VDD, GND, /Set, and Clock configuration, wire a digital input to pin 16. On pins 2 and 3, you should see /Q and Q of the D Flip-Flop reflecting the value on pin 16 delayed by one clock cycle. The Flip-Flop should be rising-edge triggered.

---

**Appendix A**

**C and Shell Script Source Code**

```c
/* FILE: codegen.c */
/* Code Generator proof-of-concept simulation */
/*
   This program simulates the xor-between-DFF style code generator by implementing a shift register in a 32-bit integer variable. Data
```
shifts through the register from right to left. When the CODEBIT is 1, all of the taps, specified by the MAGICNUM, are toggled.

To verify the codes, the program initializes the register to contain 0x00000001, and generates codes until one of the following conditions is met:

(a) the code register goes to zero --> non persisting code  
(b) the code register goes to a previous value !=1 -> non-maximal code  
(c) the code register goes back to 0x00000001 --> possibly maximal code

While it is executing a given tap sequence, it will output the set of random bits the code generates on the feedback path. It will additionally state what condition caused it to exit, and the number of states it visited before revisiting a state.

There is a special case in the code for a 31-stage code generation. Because of limitations in memory storage, it becomes difficult to track previously-visited states with register sizes much above 24-bits. So, instead, we check only for the register to come back to the all-zero, starting, or 1048576th state. Also, due to the sheer number of codes generated, none of the output bits are displayed. Further, only limited information about the closeness to maximality of a code is available, unless the code *is* maximal, in which case, it will correctly report that it is.

(This code doesn’t actually directly report if a code is maximal. Rather, it reports number of states until a code repeats. A code is maximal if it has \((2^n - 1)\) states, where \(n\) is the register size. In the case of a maximal code in the 31-stage configuration, since each state is tested for equality against the starting state, if the starting state is only reached after \((2^{31} - 1)\) steps, then the code is obviously maximal, which it will correctly find.)

*/

#include <stdio.h>

/* Pick one of THREE_STAGE, NINE_STAGE, THIRTY_ONE */

#define THREE_STAGE
#undef NINE_STAGE
#undef THIRTY_ONE

#if defined(NINE_STAGE)
#define MAGICNUM 0x021   /* 000100001 */
#define CODEBIT  0x100   /* 100000000 */
#define CODEMASK 0x1FF
#else
#if defined(THIRTY_ONE)
#define MAGICNUM 0x80000047L
#define CODEBIT  0x40000000L
#define CODEMASK 0x7FFFFFFFL
#else
#define MAGICNUM 0x3    /* 011 */
#define CODEBIT  0x4    /* 100 */
#define CODEMASK 0x7    /* 111 */
#endif
#endif

inline unsigned int nextbit (unsigned int *code)
{
unsigned int c1;

   c1=(*code)&CODEBIT;
*code=(*code<<1)&CODEMASK;
if (c1) *code ^= MAGICNUM;

   return c1!=0;
}

#define NEXTBIT(c) 
  (((c)&CODEBIT?((c)<<1)^MAGICNUM):((c)=(c<<1)&CODEMASK),0))

int visit[65536];

main()
{
  register unsigned int i,c,j,cc;

  for (i=0;i<65536;i++) visit[i]=0;

  printf("MAGIC NUMBER: %.8x\n",MAGICNUM);
  printf("CODE BIT:     %.8x\n",CODEBIT);
  printf("CODE MASK:    %.8x\n",CODEMASK);
  j=c=1;
  for (i=0;i<~(0L);i++)
  {
    /*nextbit(&c);*/
    NEXTBIT(c);
    cc=c&(CODEMASK&(~CODEBIT));
    if (c==1) break;
    #ifndef THIRTY_ONE
    if (visit[c>>4]&(1<<(c&0xf)))
    {
      printf("\nerror--revisiting point %3x at stage %d\n",c,i+1);
      exit(1);
    }
    /*visit[c>>4]|=(1<<(c&0xf));*/
    printf("%1x",!!(c&CODEBIT));
    if (i%80==79) putchar('
');
    #else
    if (j==c)
    {
      printf("\nerror--revisiting point %3x at stage %d\n",c,i+1);
      exit(1);
    }
    if (!i%1048576)
    {
      j=j>1?j:c;
    }
    #endif
    if (c==0)
    {
      printf("\nerror--code went to zero in stage %d!\n",i+1);
      exit(1);
    }
    #endif
    printf("%1x\n",!!(c&CODEBIT));
  }
printf("\n%10d stages  repeated code: %.8x\n",i+1,c);

return 0;
}

FILE: base.c */

Basic node-name extractor */

This C program runs in conjunction with build_nodelist in order to
grab labels and their node numbers into a file that can be sourced
by a Bourne shell. Once such a file is sourced, it’s possible to
use Bourne shell’s string-substitution feature to append a boilerplate
simulation to a lengthy input Spice file.* /

#include <stdio.h>
#include <string.h>
#include <ctype.h>

char *base(char *buf)
{
    char *s1,*s2;

    s1=s2=buf;
    while (s1 && *s1) { if (*s1=='/') s2=s1+1; s1++; }

    return s2;
}

main()
{
    char buf[256];
    char *s,*ss;
    int num,i;

    while (fgets(buf,255,stdin))
    {
        i=sscanf(buf,"%d",&num);
        if (i!=1) continue;
        if (strstr("FLOATING",buf)) continue;

        s=buf;
        while (s && *s) { if (*s++=='=') break; }
        while (s && *s) { if (!isspace(*s)) break; s++; }
        if (s) s=base(s);
        if (!s || !*s || !isalpha(*s)) continue;

        ss=s;

        while (ss && *ss) {
            if (*ss==':' || *ss=='.') *ss='_';
            if (!isalnum(*ss) || '_==*ss))
            {
                *ss=0;
                break;
            }
        }
        ss++;
printf("%s=%d\n",s,num);
}
return 0;
}

/* FILE: rmonte.c */
/* Resistance Monte-Carlo */
/
This program is a simple, rudimentary resistance shifter, meant to reduce the effects of perfect symmetry within the many bistable devices in my design. By shifting resistances higher and lower by small amounts within a small tolerance band, the accuracy of the simulation is not compromised, but the ability for spice to converge on a DC-bias solution is enhanced.

Note that this isn’t a cure-all as there still are many times when Spice will throw up its hands with a singular matrix. The most effective course of action that I’ve devised in that situation is to add .nodeset cards for the offending nodes to manually shift those nodes during the initial DC-bias calculation. Since these node settings become invalid at the first time point, they don’t otherwise affect the simulation as "valid" states get assigned to each bistable device.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>

/* tolerance band size, in decimal */
#define PERCENT (0.05)

double mod_resist(double d)
{
    static double blah=0.0;
    int i;
    blah=1000.0*sin(blah+1.1);
    i=blah;
    blah-=(double)i;
    if (blah<0) blah=-blah;
    if (blah>1) blah=1/blah;
    return (1.000-(PERCENT/2.0)+(PERCENT*blah))*d;
}

main()
{
    char buf[256];
    double resist;
    char *s1,*s2;
    int i;
    while (fgets(buf,255,stdin))
```

defines
{     if (buf[0]==R') /* resistor */
    
    i=0;
    s1=buf;
    while (s1 && *s1 && i<3)
    
    s1++;
    i+=isspace(s1[-1])&&!isspace(s1[0]));
    }
    
    if (s1 && *s1 && i==3) resist=atof(s1);
    else { printf("error!
"); exit(1); } 

    *s1=0;
    s2=s1+1;
    
    while (s2 && *s2)
    
    if (!(isdigit(*s2) || strchr(".+-Ee",*s2))
    
    break;
    s2++;

    } else
    
    printf("%s%f%s",buf,mod_resist(resist),s2);

    } else
    
    fputs(buf,stdout);

    return 0;
}


#!/bin/sh
#
# FILE:  build_nodelist
#
# This program builds a boilerplate simulation and appends it on the
# complete circuit description for the completed project. The resulting
# spice source can be further manually edited for performing whatever
# simulations that may be required.
#
PROJECT=Code_Generator

grep "NODE:" $PROJECT.spice | grep -v \\
| sed -e "s/\([0-9][0-9][0-9]\)/\l/g" | sed -e "s/\([0-9]\) =/\1    =/g" | sort +4 -b -d | cut -d: -f2 > $PROJECT.nodes

base < $PROJECT.nodes > $PROJECT.var
.
$PROJECT.var

# Uncomment "rmonte", and comment out the "cp" to use the monte-carlo
# resistance shifter preprocessor.
#
#rmonte < $PROJECT.spice >$PROJECT.monte

cp $PROJECT.spice $PROJECT.monte

cat header scna.spice $PROJECT.monte - << EOF | expand -4 > ${PROJECT}_a.spice

* Customized driving circuit for spice simulation of 24dx circuit in frame

EOF

```
* Power Supply
** NODE: 0  = GND!
** NODE: 1  = Vdd!

Vdd $Vdd_ = $GND_ 5v

* Clock
** NODE: $pin_36 = phi_0 input

* 100ns clock == 10Mhz
Vphi0 $pin_36 0 DC 0v PULSE(0v 5v 0ns 2ns 2ns 2ns 48ns 100ns)
* 50ns clock == 20Mhz
*Vphi0 $pin_36 0 DC 0v PULSE(0v 5v 0ns 2ns 2ns 23ns 50ns)
* 40ns clock == 25Mhz
*Vphi0 $pin_36 0 DC 0v PULSE(0v 5v 0ns 2ns 2ns 18ns 40ns)
* 20ns clock == 50Mhz
*Vphi0 $pin_36 0 DC 0v PULSE(0v 5v 0ns 2ns 2ns 8ns 20ns)

* Primary output
** NODE: $pin_01 = Qout

* Inputs
** NODE: $pin_20 = T0
*VT0 $pin_20 0 5v PULSE(5v 0v 100ns 2ns 2ns 1000s 1000s)
RT0 $pin_20 0 0.1
*RT0 $pin_20 $pin_01 0.01u
** NODE: $pin_21 = T1
RT1 $pin_21 0 0.1
*RT1 $pin_21 $pin_01 0.01u
** NODE: $pin_29 = T2
RT2 $pin_29 0 0.1
*RT2 $pin_29 $pin_01 0.01u
** NODE: $pin_22 = T3
RT3 $pin_22 0 0.1
*RT3 $pin_22 $pin_01 0.01u
** NODE: $pin_28 = T4
RT4 $pin_28 0 0.1
*RT4 $pin_28 $pin_01 0.01u
** NODE: $pin_23 = T5
RT5 $pin_23 0 0.1
*RT5 $pin_23 $pin_01 0.01u
** NODE: $pin_27 = T6
RT6 $pin_27 0 0.1
*RT6 $pin_27 $pin_01 0.01u
** NODE: $pin_24 = T7
RT7 $pin_24 0 0.1
*RT7 $pin_24 $pin_01 0.01u
** NODE: $pin_26 = T8
RT8 $pin_26 0 0.1
*RT8 $pin_26 $pin_01 0.01u
** NODE: $pin_11 = T9
RT9 $pin_11 0 0.1
*RT9 $pin_11 $pin_01 0.01u
** NODE: $pin_34 = T10
RT10 $pin_34 0 0.1
*RT10 $pin_34 $pin_01 0.01u
** NODE: $pin_12 = T11
RT11 $pin_12 0 0.1
*RT11 $pin_12 $pin_01 0.01u
** NODE: $pin_33 = T12
RT12 $pin_33 0 0.1
*RT12 $pin_33 $pin_01 0.01u
** NODE: $pin_13 = T13
RT13 $pin_13 0 0.1
*RT13 $pin_13 $pin_01 0.01u
** NODE: $pin_32 = T14
RT14 $pin_32 0 0.1
*RT14 $pin_32 $pin_01 0.01u
** NODE: $pin_14 = T15
RT15 $pin_14 0 0.1
*RT15 $pin_14 $pin_01 0.01u
** NODE: $pin_31 = T16
RT16 $pin_31 0 0.1
*RT16 $pin_31 $pin_01 0.01u
** NODE: $pin_06 = T17
RT17 $pin_06 0 0.1
*RT17 $pin_06 $pin_01 0.01u
** NODE: $pin_40 = T18
RT18 $pin_40 0 0.1
*RT18 $pin_40 $pin_01 0.01u
** NODE: $pin_07 = T19
RT19 $pin_07 0 0.1
*RT19 $pin_07 $pin_01 0.01u
** NODE: $pin_39 = T20
RT20 $pin_39 0 0.1
*RT20 $pin_39 $pin_01 0.01u
** NODE: $pin_08 = T21
RT21 $pin_08 0 0.1
*RT21 $pin_08 $pin_01 0.01u
** NODE: $pin_38 = T22
RT22 $pin_38 0 0.1
*RT22 $pin_38 $pin_01 0.01u
** NODE: $pin_09 = T23
RT23 $pin_09 0 0.1
*RT23 $pin_09 $pin_01 0.01u
** NODE: $pin_37 = T24
RT24 $pin_37 0 0.1
*RT24 $pin_37 $pin_01 0.01u

** NODE: $pin_04 = Set
VSet $pin_04 0 0v PULSE(0v 5v 10ns 2ns 2ns 100s 1000s)

* Analysis
.o option ITL1=32 ITL4=5 ABSTOL=1u RELTOL=0.005 VNTOL=5u
.tran 10ns 2500ns

.nodeset
v($Q0)=5 v($Q1)=5 v($Q2)=5 v($Q3)=5
+  v($Q4)=5 v($Q5)=5 v($Q6)=5 v($Q7)=5
+  v($Q8)=5 v($Q9)=5 v($Q10)=5 v($Q11)=5
+  v($Q12)=5 v($Q13)=5 v($Q14)=5 v($Q15)=5
+  v($Q16)=5 v($Q17)=5 v($Q18)=5 v($Q19)=5
+  v($Q20)=5 v($Q21)=5 v($Q22)=5 v($Q23)=5
+  v($Set_)=5

.control
run
plot v($Phi_1_) v($Phi_2_) v($pin_36) 5+v($Q0) 10+v($Q1) 15+v($Q2) 20+v($Q3) 25+v($Q
plot v(Phi_1) v(Phi_2) v(pin_36) 5+v(Q8) 10+v(Q9) 15+v(Q10) 20+v(Q11) 25+v(Q12) 30+v(Q13) 35+v(Q14) 40+v(Q15)
plot v(Phi_1) v(Phi_2) v(pin_36) 5+v(Q16) 10+v(Q17) 15+v(Q18) 20+v(Q19) 25+v(Q20)
rusage all
.endc
EOF