

DIY Calculator Demo: Pseudo-Random Number Generator

Introduction

The purpose of this paper is to provide a brief introduction to the virtual computer-calculator known as the DIY Calculator that accompanies our book *How Computers Do Math* (ISBN: 0471732788). You can download a fully-functional free copy of the software from our website at www.DIYCalculator.com (you'll find instructions on how to download and install the calculator on the website).

A Quick Refresher: Computers and Calculators

In its broadest sense, a computer is a device that can accept information from the outside world, process that information using logical and/or mathematical operations, make decisions based on the results of this processing, and ultimately return the processed information to the outside world in its new form.

The main elements forming a computer system are its central processing unit (CPU), the memory devices (ROM and RAM) that are used to store programs (sequences of instructions) and data, and the input/output (I/O) ports that are used to communicate with the outside world. The CPU is the "brain" of the computer, because this is where all of the number-crunching and decision-making is performed.

Once we've conceived the idea of a general-purpose computer, the next step is to think of something to do with it. In fact there are millions of tasks to which computers can be assigned, but the application we're interested in here is that of a simple calculator. So what does it take to coerce a computer to adopt the role of a calculator? Well, one thing we require is some form of user interface, which will allow us to present data to, and view results from, the computer (Figure 1).

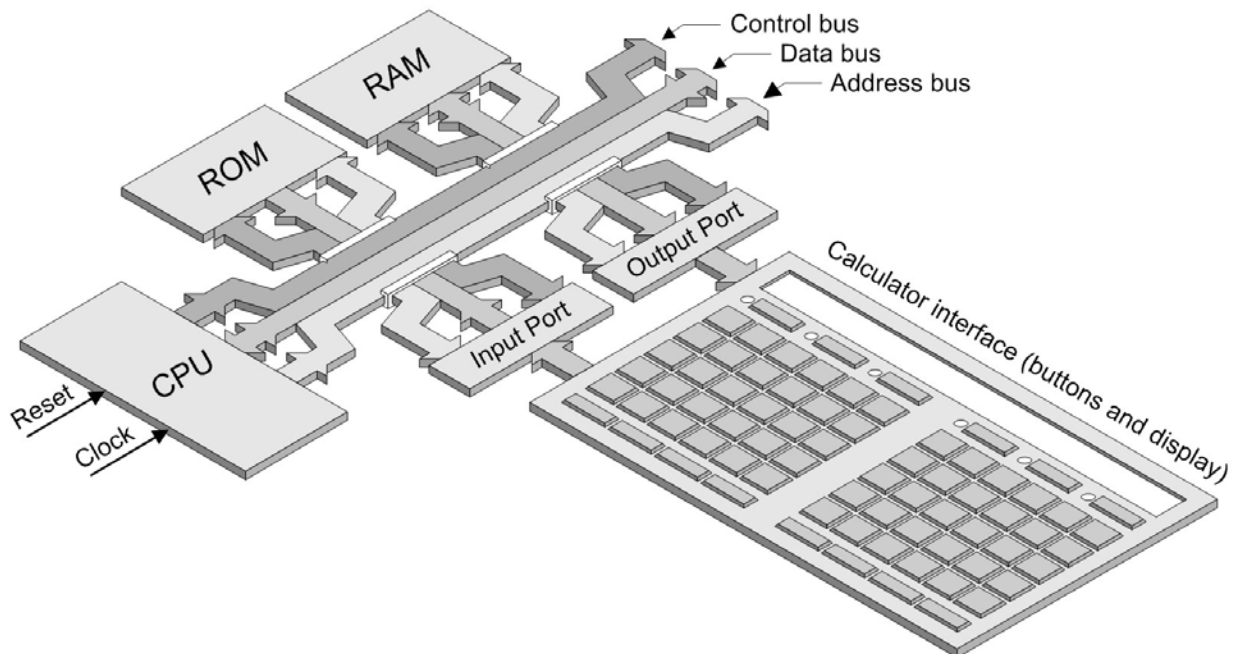


Figure 1. The main elements forming a computer-calculator

(This illustration was reproduced from *How Computers Do Math* with the kind permission of the publisher)

The calculator's user interface primarily consists of buttons and some type of display. Each button has a unique binary code associated with it, and this code will be presented to the computer's input port whenever that button is pressed. Meanwhile, one of the computer's output ports can be used to drive the display portion of the interface.

A Simple Test Case

In order to provide a simple demonstration as to how the DIY Calculator works, download your free copy and install it as described on our website. Now use the **Start > Programs > DIY Calculator > DIY Calculator** command (or double-click the **DIY Calculator** icon on your desktop) to launch this little rapsallion, which should look something like the screenshot shown in Figure 2.

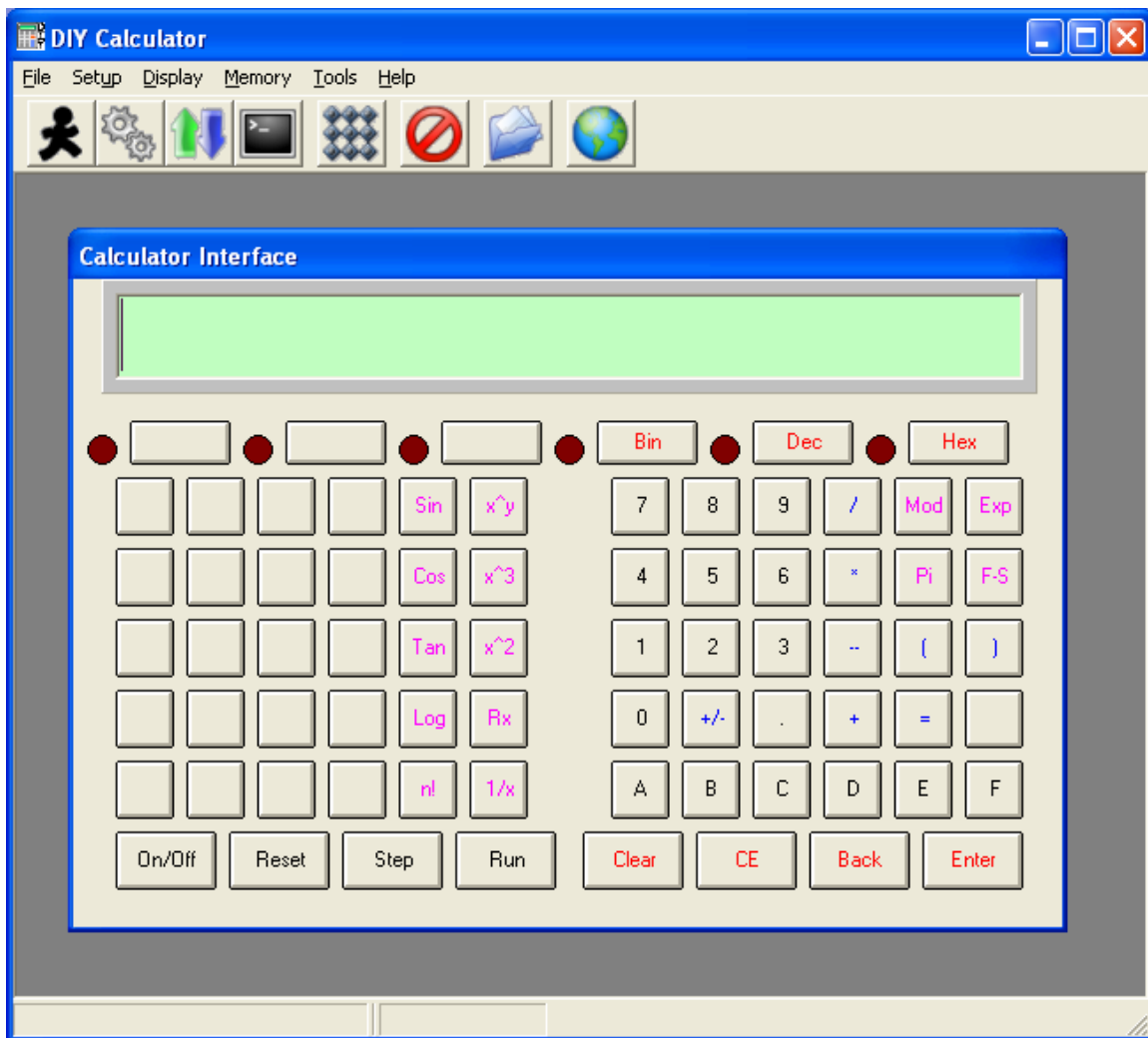


Figure 2. Screenshot of the DIY Calculator's interface

Click the **On/Off** button on the calculator interface to power it up and then click on some of the '0' through '9' buttons. Nothing happens, because we haven't loaded a program yet. In a

moment we are going to create our own program, but just to provide a simple example, we've already provided a test case for you to play with as part of the download.

Use the **Tools > Assembler** command (or click the appropriate icon in the main window's tool bar) to launch an application called the assembler. Now use the assembler's **File > Open** command to open the file called *hello.asm* that you'll find in the *C:\DIY Calculator\Work* folder on your system.

This program is in a low-level language known as assembly language. Each computer has its own assembly language, but once you've learned one (especially one as simple as ours), this makes it much easier to learn others if you need to do so.

Now use the assembler's **File > Assemble** command. This generates a new file called *hello.ram* that contains the raw instruction and data values that will be processed by our virtual CPU. The contents of this file are in a form called "machine code," because this is the form that is actually executed by the computer (machine).

Use the assembler's **File > Exit** command to dismiss this application. Next, use the **Memory > Load RAM** command to load the *hello.ram* file that you'll find in the *C:\DIY Calculator\Work* folder into the DIY Calculator's memory. Finally, click the **Run** button to execute this program and see the message "Hello World" appear on the calculator's main display.

A Pseudo-Random Number Generator

In order to get a better feel for how the DIY Calculator performs its magic, we are going to create our own program from the ground up. The program we've chosen to implement is one that will generate a sequence of pseudo-random numbers in the range 0 through 7. Furthermore, we're going to include the ability to specify a "seed" value to act as a starting point.

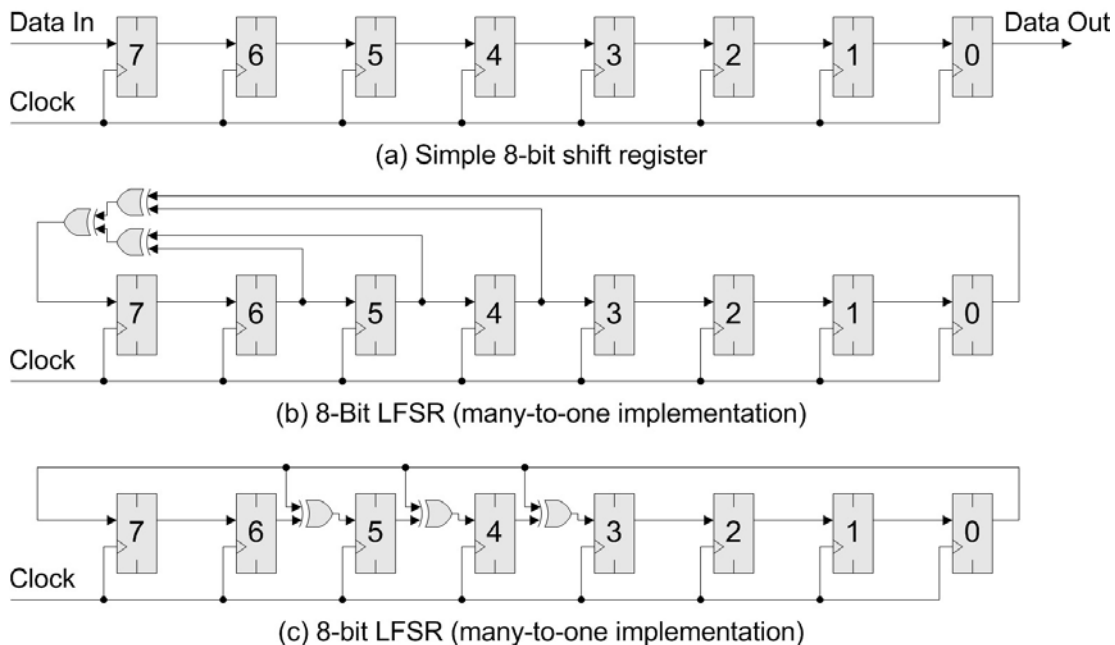


Figure 3. Simple shift register and two different LFSR implementations

There are many different techniques for generating pseudo-random numbers. One of the simplest methods is to use the concept of a *Linear Feedback Shift Register (LFSR)*. First, consider a simple 8-bit shift register in which the register bits are numbered from 7 to 0 as shown in Figure 3(a). Each register (memory element) can store a binary 0 or 1 value, and all eight register elements are driven by a common `Clock` signal. When a rising (0-to-1) transition is presented to the `Clock` signal, the value on the `Data In` signal is copied into register 7; at the same time, the current contents of register 7 are copied into register 6; the current contents of register 6 are copied into register 5; and so on down the line.

By comparison, let's now consider the LFSR shown in Figure 3(b). In this case, the input to the register is formed by XOR-ing feedback signals from a number of points called "taps" in the register chain. Now an 8-bit register can contain $2^8 = 256$ different patterns of 0s and 1s. If you select the right taps (which we have of course), you end up with a "maximal length LFSR" that will pseudo-randomly pass through every possible combination of values (except all 0s) before returning to its starting point.

The LFSR in Figure 3(b) is known as a "many-to-one" implementation, because multiple taps are fed back to the input. A complementary form of LFSR, called a "one-to-many" implementation, takes a single tap – the output from bit 0 – and feeds it back via XOR gates into multiple points in the register chain as illustrated in Figure 3(c). It is this latter form that is of interest to us here, because this type is more easily implemented in software.

The Accumulator (ACC) and Status Register (SR)

There are just a couple more things we need to know before we plunge headfirst into the fray. As illustrated in Figure 4, amongst other things, our CPU contains two 8-bit registers called the *accumulator (ACC)* and the *status register (SR)*. (In this context, the term "register" refers to a group of memory elements, each of which can store a single binary digit.)

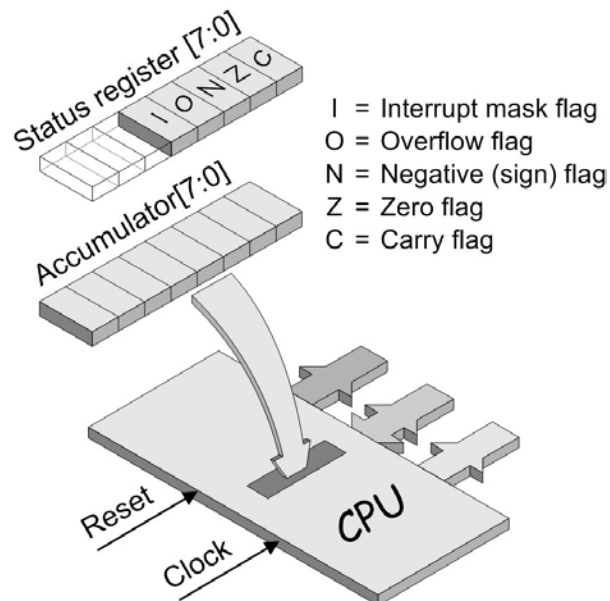


Figure 4. The accumulator (ACC) and status register (SR)

As its name implies, the accumulator is where the CPU gathers, or "accumulates", intermediate results. In the case of the status register, each of its bits is called a *status bit*, but they are also

commonly referred to as *status flags* or *condition codes*, because they serve to signal (flag) that certain conditions have occurred. We will concern ourselves only with the carry (C) and zero (Z) flags for the purposes of our example program.

Since we may sometimes wish to load the status register from (or store it to) the memory, it is usual to regard this register as being the same width as the data bus (8 bits in the case of our system). However, our CPU employs only five status flags, which occupy the five least-significant bits of the status register. This means that the three most-significant bits of the register exist only in our imaginations, so their non-existent contents are, by definition, undefined.

Binary and Hexadecimal

Last but not least, computers internally store and manipulate data using the binary number system, which comprises just two digits: 0 and 1. One wire (or register bit / storage element) can be used to represent two distinct binary values: 0 or 1; two wires can represent four binary values: 00, 01, 10, and 11; three wires can represent eight binary values: 000, 001, 010, 011, 100, 101, 110, and 111; and so on. As our virtual computer has an 8-bit accumulator, this can store 256 different binary values numbered from 0 to 255 in decimal or %00000000 to %11111111 in binary (where the “%” symbol is used to indicate a binary value).

The problem is that humans tend to find it difficult to think in terms of long strings of 0s or 1s. Thus, when working with computers we tend to prefer the hexadecimal number system, which comprises 16 digits: 0 through 9 and A through F as shown in Figure 5.

Value	Decimal	Binary	Hexadecimal
zero	0	%0000	\$0
one	1	%0001	\$1
two	2	%0010	\$2
three	3	%0011	\$3
four	4	%0100	\$4
five	5	%0101	\$5
six	6	%0110	\$6
seven	7	%0111	\$7
eight	8	%1000	\$8
nine	9	%1001	\$9
ten	10	%1010	\$A
eleven	11	%1011	\$B
twelve	12	%1100	\$C
thirteen	13	%1101	\$D
fourteen	14	%1110	\$E
fifteen	15	%1111	\$F

Figure 5. Binary and hexadecimal

Note that we use “%” and “\$” characters to indicate binary and hexadecimal values, respectively. Each hexadecimal digit directly maps onto four binary digits (and vice versa of course).

The Address, Control, and Data Busses

The term “bus” is used to refer to a group of signals that carry similar information and perform a common function. A computer actually makes use of three buses called the control bus, address bus, and data bus.

The CPU uses its address bus to “point” to other components in the system; it uses the control bus to indicate whether it wishes to “talk” (output / write / transmit data) or “listen” (input / read / receive data); and it uses the data bus to convey information back and forth between itself and the other components.

Our virtual computer is equipped with a data bus that is 8 bits wide and an address bus that is 16 bits wide. This allows the address bus to point to $2^{16} = 65,536$ different memory locations, which are numbered from 0 to 65,535 in decimal; %0000000000000000 to %1111111111111111 in binary; or \$0000 to \$FFFF in hexadecimal.

The Program Itself

In a moment we’re going to enter the program shown below, but before we do that, let’s walk through the code step-by-step.

```
##### Generate pseudo-random numbers between 0 and 7

##### Declare constants
MAINDISP: .EQU    $F031    # Output port for main display
KEYPAD:    .EQU    $F011    # Input port for keypad

CLRCODE:   .EQU    $10      # Code to clear the main display
QCODE:     .EQU    $3F      # ASCII code for "?" (Query)
CCODE:     .EQU    $3A      # ASCII code for ":" (Colon)
ENTERKEY:  .EQU    $13      # Code associated with "Enter" key

xx000xxx:  .EQU    %00000000 # Mask value (XOR with 0s)
xx111xxx:  .EQU    %11111111 # Mask value (XOR with 1s)

                .ORG      $4000    # Set program origin

##### Clear main display and show a "?"
INIT:         LDA      CLRCODE    # Load accumulator with clear code
              STA      [MAINDISP] # Copy to main display
              LDA      QCODE      # Load accumulator with "?"
              STA      [MAINDISP] # Copy to main display

##### Get a seed value, store it, and display it
GETSEED:     LDA      [KEYPAD]    # Load accumulator from the keypad
              CMPA     $0F        # Compare accumulator to code $0F
              JC       [GETSEED]  # Jump back if C flag is set

DISPSEED:    STA      [LFSR]      # Otherwise copy to LFSR
              LDA      CLRCODE    # Load accumulator with clear code
              STA      [MAINDISP] # Copy to main display
              LDA      [LFSR]      # Load accumulator with seed value
              STA      [MAINDISP] # Copy to main display
              LDA      CCODE      # Load accumulator with ":"
              STA      [MAINDISP] # Copy to main display
```

```

##### Generate a random number and display it
##### Wait for "Enter" key
WAITENT:  LDA      [KEYPAD]    # Load accumulator from the keypad
          CMPA    ENTERKEY    # Compare to code for "Emter" key
          JNZ     [WAITENT]   # Jump if not the right code

##### Generate the new random value and display it
GENRAND:  LDA      [LFSR]     # Load accumulator with LFSR
          SHR     # Shift right 1 bit

TEST0OR1: JC      [ITS1]     # Jump to XOR with 1

ITS0:     AND     %01111111   # Clear MS bit to 0
          STA     [LFSR]     # Store value in LFSR
          XOR     xx000xxx    # XOR bits [5,4,3] with 0
          AND     %00111000   # Mask out bits [7,6,2,1,0]
          STA     [TEMP]     # Store it
          JMP     [MERGE]    # Jump to "Merge" bits together

ITS1:     OR      %10000000   # Set MS bit to 1
          STA     [LFSR]     # Store value in LFSR
          XOR     xx111xxx    # XOR bits [5,4,3] with 1
          AND     %00111000   # Mask out bits [7,6,2,1,0]
          STA     [TEMP]     # Store it

MERGE:    LDA     [LFSR]     # Load accumulator with LFSR
          AND     %11000111   # Mask out bits [5,4,3]
          OR      [TEMP]     # Bring in the XORed bits
          STA     [LFSR]     # Store in LFSR
          AND     %00000111   # Mask out LS 3 bits
          STA     [MAINDISP]  # Copy to main display
          JMP     [WAITENT]   # Jump back and do it again

LFSR:     .BYTE # The LFSR itself
TEMP:     .BYTE # A temp storage location

          .END # This is the end of the program

```

The first thing we do is to declare some labels and associated them with certain values using `.EQU` ("equate") commands. For example, the `MAINDISP` label is associated with the hexadecimal value `$F031`, which is the address of the output port that drives the calculator's main display. Similarly, the `KEYPAD` label is associated with the hexadecimal value `$F011`, which is the address of the input port that is connected to the calculator's keypad.

Note that everything to the right of a `#` character is treated as a comment and is therefore ignored by the assembler.

Now, before we go any further, take a moment to look at the end of the listing, where we see two `.BYTE` commands associated with the labels `LFSR` and `TEMP`. Each of these commands reserves a byte (an 8-bit value) in which we can store intermediate values.

OK, let's return to the top of the program. Following the `.EQU` commands we see a `.ORG` ("origin") statement, which we use to specify `$4000` as being the first address in our program. (The reason we use `$4000` is that this is the first address in the DIY Calculator's virtual RAM).

INIT: When we get to the `INIT` (“initialize”) label, we use a `LDA` (“load accumulator”) instruction to load the accumulator with a special code called `CLRCODE`. Then we use a `STA` (“store accumulator”) instruction to copy this code to the main display, thereby clearing the display.

Next, we load the accumulator with a code called `QCODE` that corresponds to the question mark “?” character and write that code to the display. Note that `CLRCODE` and `QCODE` are labels to which we previously assigned values at the beginning of the program. (The origin of these code values and their use is explained in our book. For our purposes here, we need only note that a “?” character will now appear on the calculator’s main display.)

GETSEED: We now find ourselves at the `GETSEED` label, where we are going to wait for the user to enter a seed value in the form of a hexadecimal digit in the range \$1 to \$F. The user will do this by clicking on one of the ‘1’ through ‘9’ or ‘A’ through ‘F’ buttons on the calculator’s keypad. (We don’t wish the user to click the ‘0’ key, thereby entering a value of \$0, because this is an illegal value for the type of LFSR we are creating. If our LFSR were to be loaded with all zeros, it would never be able to get out of this state. Having said this, we are not going to perform any error checks to ensure that the user doesn’t click the ‘0’ key in this simple program.)

Now, one thing we need to know is that our virtual calculator’s front panel contains an 8-bit storage element called a latch. By default, this latch is loaded with a value of \$FF. When we click a button on the keypad, a code associated with that button is loaded into the latch. When the CPU reads from the input port connected to the calculator’s keypad, it actually reads the value out of this latch; furthermore, the act of performing this read automatically reloads the latch with its default \$FF value. Last but not least, the hexadecimal codes associated with the ‘0’ through ‘9’ and ‘A’ through ‘F’ keys are \$00 through \$09 and \$0A through \$0F, respectively (we’d have been extremely silly to make this work any other way).

Thus, when we arrive at the `GETSEED` label, we load the accumulator from the memory location pointed to by the `KEYPAD` label, which is the address of the input port connected to the calculator’s keypad. Next, we use a `CMPA` (“compare accumulator”) instruction to compare the contents of the accumulator to \$0F. If the value in the accumulator is the larger, the carry (C) status flag will be loaded with 1; this means that the user clicked a button whose code is higher than \$0F, which we don’t wish to happen. Thus, if the user did click a button other than ‘1’ through ‘9’ or ‘A’ through ‘F’, the `JC` (“jump if carry”) instruction will return the program to the `GETSEED` label to await another key.

DISPSEED: Alternatively, if the user did click a button between ‘1’ and ‘F’, the `JC` will fail and the program will continue to the `DISPSEED` (“display seed”) label. When we reach this label, we use a `STA` (“store accumulator”) instruction to copy the seed value into the temporary `LFSR` location we reserved at the end of the program. Then we load the accumulator with the `CLRCODE` value and copy this to the main display to clear it. Next, we reload the accumulator with our seed value and copy it to the main display so that we can see a confirmation of the key we clicked. Now we load the accumulator with a code called `CCODE` that corresponds to the colon “:” character and write that code to the display. This character will allow us to distinguish between the seed value and any random numbers we generate in the future. (Note once again that `CODE` is a label to which we previously assigned a value at the beginning of the program.)

WAITENT: We now find ourselves at the `WAITENT` (“wait for the enter key”) label. The first thing we do here is to wait for the user to click on the “Enter” key. In order to do this, we load the accumulator with a value from the keypad and compare it with the code for the “Enter” key (we

do this using the `ENTERKEY` value we declared at the beginning of the program). If the user hasn't clicked the "Enter" key, this comparison will result in the zero (Z) flag containing 0, in which case the `JNZ` ("jump if not zero") instruction will return us to the `GENRAD` label to await the real key to be pressed. (The way the zero flag and the `JNZ` instruction works may seem a little counter-intuitive at first, but it does make sense once you realize how the status flags work. As you might expect, this is discussed in excruciating detail in our book, *How Computers Do Math*.)

GENRAND: Once the user does click the Enter" key, we find ourselves at the `GENRAND` ("generate random number") label. This is where things start to get interesting. First, we use a `SHR` ("shift right") shift the contents of the accumulator one bit to the right. This means that the original contents of bit 7 are copied into bit 6; the original contents of bit 6 are copied into bit 5; and so on down the line. Of particular interest is the fact that the original contents of bit 0 conceptually "fall off the end" and are copied into the carry (C) flag.

We should also note that the DIY Calculator's `SHR` instruction performs an arithmetic shift right, which means that the original contents of bit 7 are copied back on itself. However, this isn't important because we are going to overwrite this bit as discussed below.

TEST0OR1: Remembering that, following the shift right discussed above, the original contents of bit 0 (the bit that "fell off the end") are now contained in the carry (C) flag, the `JC` ("jump if carry") instruction at the `TEST0OR1` ("test if the carry flag contains 0 or 1") label is used to determine if this bit is a 0 or 1. If it's a 1, the `JC` will pass and we will jump to the `ITS1` ("it's a 1") label; otherwise the test will fail, and we'll end up at the `IST0` ("it's a 0") label.

ITS0: Return to Figure 3(c) for a moment and observe that the output from bit 0 of the linear feedback shift register is copied back to the input of bit 7. From the test discussed above, we know that this bit was a 0, so the first thing we do at the `ITS0` ("it's a 0") label is to use a logical `AND` instruction to clear the most-significant bit of the accumulator to 0. Next, we copy the value in the accumulator back into our `LFSR` location.

Now look at the three `XOR` gates shown in Figure 3(c). In the case of our software realization, we've already performed the shift, so we know that the original contents of bit 6 are now in bit 5; the original contents of bit 5 are in bit 4, and the original contents of bit 4 are in bit 3. We also know that the bit that fell off the end is a 0, so we use the `XOR` instruction to `XOR` bits 5, 4, and 3 of the accumulator with 0s. At this time we aren't concerned with the other bits, so we use an `AND` instruction to clear bits 7, 6, 2, 1 and 0, and we store the result in our `TEMP` temporary location. Finally, we jump to the `MERGE` label to bring everything back together as discussed below.

ITS1: As we know, we arrive at this point in the program if the bit that fell of the end following out shift was a 1. In this case, the first thing we do at the `ITS1` ("it's a 1") label is to use a logical `OR` instruction to set the most-significant bit of the accumulator to 1. Next, we copy the value in the accumulator back into our `LFSR` location.

Once again, consider the three `XOR` gates shown in Figure 3(c). As before, we've already performed the shift, so we know that the original contents of bit 6 are now in bit 5; the original contents of bit 5 are in bit 4, and the original contents of bit 4 are in bit 3. We also know that the bit that fell of the end is a 1, so we use the `XOR` instruction to `XOR` bits 5, 4, and 3 of the accumulator with 1s. Again, we aren't concerned with the other bits at this time, so we use an

AND instruction to clear bits 7, 6, 2, 1 and 0, and we store the result in our TEMP temporary location.

MERGE: Finally, we arrive at the MERGE label, where we are going to bring everything back together. First we load the accumulator with the contents of our LFSR location. We know that we wish to replace the contents of bits 5, 4, and 3, so we use an AND instruction to clear these three bits to 0. Next, we use an OR instruction to merge in bits 5, 4, and 3 from our temporary location TEMP and we squirrel a copy of the result back into our LFSR location for use when generating the next value.

Now we use another AND instruction to mask out everything except the three least-significant bits in the accumulator, and we copy the result (which will be a number between 0 and 3) to the main display. Finally, we jump back to the WAITENT label to await the user clicking the “Enter” key to generate the next pseudo-random number in the sequence.

Entering and Running the Program

Now we’re really ready to rock and roll. If you haven’t already done so, launch the DIY Calculator and invoke the assembler. (If you still have the assembler open from running the test case earlier, then use its **File > New** command to create a new source code file.)

Enter the program shown above, use the assembler’s **File > Save As** command to save this program with the name *lfsr.asm*, and then use the assembler’s **File > Assemble** command to translate your source program into a machine code equivalent called *lfsr.ram* (if any errors are reported in the status bar at the bottom of the assembler window, debug them and re-assemble the program). Finally, use the assembler’s **File > Exit** command to dismiss this application.

Click the **On/Off** button on the calculator interface to power it up. (Alternatively, if the calculator is still powered up from running the test case earlier, then use the main window’s **Memory > Purge RAM** command to delete the old program from the calculator’s memory.) Next, use the **Memory > Load RAM** command to load the *lfsr.ram* file that you just created into the DIY Calculator’s memory.

Now, click the **Run** button to execute this program and observe that the main display is cleared and a question mark appears. The program is now looping around waiting for you to press a key. Click a hexadecimal number between ‘1’ and ‘9’ or ‘A’ to ‘F’ and observe that the display is again cleared and this number (your “seed” value) appears followed by a colon “:” character.

Now click the “Enter” key a few times and observe the ensuing sequence of pseudo-random numbers between 0 and 7. Copy this sequence down onto a piece of paper and then click the **Reset** button to reset the calculator and the **Run** button to re-run the program. Once again, the program is looping around waiting for you to press a key. Click another hexadecimal number between ‘1’ and ‘9’ or ‘A’ to ‘F’ to enter a different seed value and then click the “Enter” key a few times. Compare this pseudo-random sequence to the previous one.

But Wait, There’s More!

This simple demonstration has only scratched the surface of what is possible with the DIY Calculator. For example, click the **Reset** button on the calculator’s front panel. Make sure the main window fills your screen, and then use the **Display > CPU Registers**, **Display > Memory**

Walker, and **Display > I/O Ports** commands to invoke these utilities. As each tool appears, drag it to a clear area on your screen (if you have enough room on your screen, you might also try launching the **Display > Message System** utility).

Now click the **Step** button on the calculator's front panel a few times and watch what happens in the various displays for each click. Next, click one of the number buttons – say the '6' key – and then click the **Step** button a couple more times, again watching the various displays for each click.

Note that you can use the main window's **Help > Contents** command to learn more about what these diagnostic tools do.

Further Projects

In our book, *How Computers Do Math*, we have a series of chapters introducing fundamental concepts such as "Subroutines" and "Recursion." Each chapter is backed up by a suite of interactive laboratories, each of which details what the reader will learn and how long it will take (typically 20 to 40 minutes), followed by step-by-step instructions that walk the reader through that lab.

For educators, the CD ROM accompanying the book includes all of the labs as Adobe® Acrobat® files that can be printed out and used as handouts. Also, all of the illustrations in the book are provided as PowerPoint® slides that can be used as the basis for presentations.

The chapters and labs build on each other until, at the end, we have a four function calculator that can input numbers in decimal, convert them into 16-bit binary integers, perform addition, subtraction, multiplication, and division on these binary values, and then present the results from these calculations in decimal on the main display.

But this is only a starting point. On the DIY Calculator's website, we are going to develop this much further by introducing the concept of floating-point values, describing our own simple floating-point format, and then implementing binary floating-point versions of our input, output, and math subroutines. [We are also going to do the same for binary Coded decimal (BCD) – check the website for more details.]

And this is still just the beginning, because (in conjunction with schools, colleges, universities, and individual readers), we plan on creating subroutines (with associated documentation) to implement many more math functions. These will be provided on the website for folks to download and experiment with and – hopefully – to say "*I can do better than that*" and send in their own versions.

As yet another example of something that may interest educators, as a final year project at the beginning of 2005, a team of students at the University of Newcastle upon Tyne, UK, created VHDL models of the *DIY calculator* and then implemented a physical version of the little scamp using a field-programmable gate array (FPGA) development board. (Details of this project, including the VHDL source files and the project notes, are available on our website.)

So there you have it. Now there is a way to learn about how computers work – and how they do math – that's actually fun and doesn't make your brain ache or your eyes water. We'd love to hear what you think about all of this, so please feel free to email us at info@DIYCalculator.com to share your thoughts and ideas.