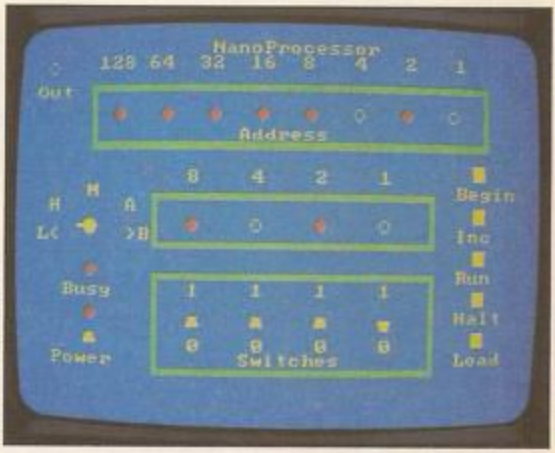
# The NanoProcessor

**by Roger Wood**
**and**
**Wayne Koberstein**
*HCM Staff*

***With this simple simulation of the machine's inner workings, you can discover how easy (and fun!) it is to communicate with computers in their own language.***




Since the premiere of the movie *Tron*—in which the hero has to fight his way out of a computer's microcircuits—many people have held a fascination for the inner workings of this "thinking machine." Are you one of them? Perhaps your interest has always been there, but you have not yet "taken the plunge" into machine-level programming. Or perhaps you know a great deal about this subject already, but would appreciate a very clear and simple demonstration of how computers "think." If so, you're ready for *NanoProcessor*—a program that emulates the computer at its most fundamental level.

At the heart of a computer, there is nothing but an immense set of on and off switches. But how can such a simple foundation foster such a complex information-handling system? In short, how are all these switches organized? A "real" computer, such as the one you have at home, is such a large system that it would be difficult to see the forest for the trees. But, with *NanoProcessor*, you have a chance to operate and see a much-simplified model of how a computer performs its tasks.

## Brain Central

All computers—including the *NanoProcessor*—have a central "brain." It's called the CPU (Central Processing Unit). This brain recognizes and responds to different sets of numbers as instructions. These instructions direct the CPU to carry out certain operations—much as our brains store, handle, and act on information encoded in switch-like neurons. In a computer, information travels along parallel paths of wires and printed circuits called "buses."

As humans, we may think in English, Spanish, or any other language—some subtle, some exact. Computers also "think" in languages—such as BASIC and LOGO. CPU's like our own brains, must translate these high-level languages into encoded information. In computers, this information takes the form of machine language—a set of codes and numerical values expressed as binary numbers. Binary means "two," and implies two choices: on or off; or, in purely numerical terms, 1 or 0.

People tend to think in terms of a ten-based number system because they have ten fingers—but a switch has only two "fingers." (For a detailed look at converting between these number systems see the sidebar "Numbers To Bits And Back.") When you `RUN` *NanoProcessor* you will notice the row of switches at the bottom of the screen—your only means of shuttling information through this simulated computer (See Photos 1, 2, 3). Each switch only has two positions—up for *on* (1), or down for *off* (0). A switch is therefore the perfect means for conveying binary information.

## Banking on Memory

Every computer has a memory area, called "Random Access Memory" (RAM), and a Central Processing Unit (CPU). Memory is the computer's capacity to store information, and is measured in terms of "bytes." A byte generally consists of 8 *bits* of information—where a bit is one binary (on or off) condition.

A CPU performs all the arithmetic that manipulates the numerically-encoded data—the ones and zeros—stored in a computer's RAM. This memory is made up of discrete "locations" in the machine, each of which has an "address." It helps to think of each memory location as a mailbox that not only has an address attached to it, but also a place to put the mail. This mail is the data stored at that location. Each "mailbox" has a limited amount of space that depends on the machine design. Because each of *NanoProcessor*'s memory locations can only store 4 bits, (one nibble), we say it is "nibble-addressable." By simply requesting a particular address, the CPU can immediately find what is contained at that address. This direct addressability of memory by the CPU is what gives a computer the power of *random access*.

The CPU and RAM are connected by three buses: the address bus (8 parallel wires), the data bus (4 parallel wires), and the control bus (See Figure 2). The first provides access to each memory location; the second simply moves data to and from each location; and the third carries control signals which control the flow of data between the CPU and memory. Furthermore, the CPU is organized into a system of discrete "registers" that serve as temporary stations for storing and shuffling data.

Look at the *NanoProcessor* front panel. On the middle-left side of the screen is a "rotary switch" with various letters positioned around it. The letters on the right-hand side of this switch—A and B—stand for the A and B registers in the CPU. It is between these two registers that the actual "arithmetic" and logic operations take place. The A register is also called the Accumulator because this is where the answers to many of the commands end up—or *accumulate*.

SOFTWARE INSTRUCTIONS

## NUMBERS TO BITS AND BACK

One of the most important aspects of machine language programming (but sometimes most confusing for the novice) is converting digital numbers to binary and vice versa. To make this as easy as possible, we have employed two aids: 1) Whenever we list a *binary* number, we precede it with a percent (%) sign; and 2) *NanoProcessor* displays the decimal equivalent of each bit above the address and data windows of the front panel (see diagram below). We refer to these decimal equivalents as the "weight" of the bits.

To quickly convert a binary number to a decimal number, simply add up the weights of the "1" (on) bits. For example, to convert %1111 1010, refer to the following diagram:

| | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|
| | * | * | * | * | * | * | * | * |
| % | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |

Then add 128+64+32+16+8+2 and you can easily arrive at the correct decimal equivalent: 250. (Also, see Figure 1 for converting the numbers 0—15 to binary.)

**Figure 1**

| Decimal | Binary |
|---|---|
| 0 | %0000 |
| 1 | %0001 |
| 2 | %0010 |
| 3 | %0011 |
| 4 | %0100 |
| 5 | %0101 |
| 6 | %0110 |
| 7 | %0111 |
| 8 | %1000 |
| 9 | %1001 |
| 10 | %1010 |
| 11 | %1011 |
| 12 | %1100 |
| 13 | %1101 |
| 14 | %1110 |
| 15 | %1111 |

### Turning On

First, press **P** to turn on the Power to your *NanoProcessor*. Make sure the rotary switch is pointing to the letter M, for Memory. You move this switch left (counter-clockwise)with the < (less than) key, and right (clockwise) with the > (greater than) key.

At the top of the screen, you should see an address box containing a long row of "lights" with numbers across the top. This is the "location counter" shown inside the CPU of Figure 2. It displays the 8-bit address of the location currently being interrogated by the CPU. Notice the vertical row of buttons at the right side of the screen. These buttons represent *NanoProcessor*'s functions. Press the **B** (for **B**egin) key on your keyboard. This effectively turns off all the lights in the address box, indicating that you have returned to the first address in memory: the 0 (zero) location. Now press the **I** key, for Increment. This moves you to the next address: location 1. If you repeatedly press **I**, you will continue to step through successive locations.

Notice that, as you step through each location, the row of 8 lights in the address box changes. These lights display the *address* of the "mailbox." To view the *contents* of this mailbox, look at the row of 4 lights directly above the toggle switches. This shows the value stored at the current location. If you were to move the rotary switch pointer to A, you would see the contents of the A register. To examine the B register, point the switch to the letter B. Now, move the pointer to the letters H or L at left. These access the "high nibble" (the first or left-most 4 bits) and the "low nibble" (the last or right-most 4 bits) in the 8-bit address.

### Entering Data

The next step is to "fill" these locations so that the processor has something to process. With the rotary switch in the M position, try toggling the switches in the switch box. Nothing happens? Don't worry; turn some of these switches "up" and then press **L**, for **L**oad. Now you have something. Any switch that is *on* has a corresponding light glowing just above it.

You have just entered your first "data" into the *NanoProcessor*. Now move the rotary switch to the H position and try the same exercise. This time, when you press **L**, lights not only come on in the "contents" box, but the same pattern of lights appears in the high (left-most) nibble of the address box. Moving the rotary switch to L (for Low nibble) and loading a value affects the low nibble (right-most) half of the 8-bit address in the same way. Once you have thus designated a full 8-bit address, move the pointer to the M position again to view the contents of *that same address*. By doing this, you have, in effect, moved to this address location, and can enter data there.

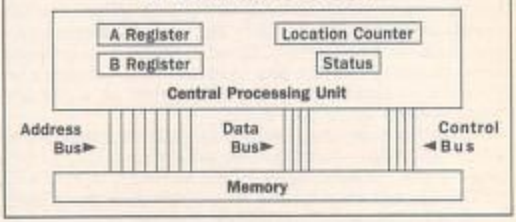
If you next move the rotary switch pointer to the A or B position and try to enter data, you will not be able to—because whatever goes in or out of these registers has to do so while the *NanoProcessor* is running instructions encoded into memory. You will also notice a small Output light (labeled "Out") at the upper left of the screen. We will explain the use of this in the *NanoAssembler* program next issue.

Your next job is to enter your first machine-language program on the *NanoProcessor*.

### Programming The Machine

A CPU executes commands *sequentially*. As it runs a program, it steps through this sequence in much the same way you "incremented" through each memory location. However, the program may instruct the CPU to take other paths—"branching" to many different locations before completing its task. You are able to program this processor by entering three different kinds of data: 1) encoded commands; 2) pure numbers; and 3) addresses. As with any program, it is the *logic* of this sequence that determines what the processor will do.

**Figure 2**
**Simplified Block Diagram of the NanoProcessor**




*NanoProcessor* understands 16 different commands—its "instruction set." Although initially expressed in one nibble, some commands require additional memory locations to hold the data necessary to execute the command. Figure 3 lists these 16 commands, showing each corresponding binary code; how many nibbles in a program the instruction requires; its "mnemonic"; which (if any) flags in the status register the instruction affects; and a brief explanation of the command function. As you develop more complicated programs, you will have to understand and use more of these commands. But, for now, try a very short routine—one that simply adds two small numbers together.

## Figure 3: Instructions Set

| Dec. | Binary | Nibbles per instr. | Mnemonic | Flags* affected C Z | Function |
|---|---|---|---|---|---|
| 0 | %0000 | 1 | ADD | Y Y | Add the contents of B register to the contents of A register—result in A. |
| 1 | %0001 | 2 | LDA # | N Y | Load A with number following instruction. |
| 2 | %0010 | 3 | LDA addr | N Y | Load A with number at location specified by addr. |
| 3 | %0011 | 3 | STA addr | N N | Store the contents of A at location specified by addr. |
| 4 | %0100 | 1 | TAB | N N | Transfer contents of A to B. |
| 5 | %0101 | 1 | TBA | N Y | Transfer contents of B to A. |
| 6 | %0110 | 1 | RRC | Y Y | Rotate A right one bit through carry. |
| 7 | %0111 | 1 | RLC | Y Y | Rotate A left one bit through carry. |
| 8 | %1000 | 1 | AND | Y Y | Logically AND A and B—Result in A. |
| 9 | %1001 | 1 | OR | Y Y | Logically OR A and B—Result in A. |
| 10 | %1010 | 1 | XOR | Y Y | Logically XOR A and B—Result in A. |
| 11 | %1011 | 3 | BZ addr | N N | Branch to addr if Zero flag is set. |
| 12 | %1100 | 3 | BNZ addr | N N | Branch to addr if Zero flag is not set. |
| 13 | %1101 | 3 | BCS addr | N N | Branch to addr if Carry flag is set. |
| 14 | %1110 | 3 | BCC addr | N N | Branch to addr if Carry flag is not set. |
| 15 | %1111 | 3 | JMP addr | N N | Branch to addr unconditionally. |

**Flags affected refers to whether or not the instruction has any effect on the flags in the status register. The C column stands for the Carry flag (did the operation result in a carry being generated?), and the Z stands for the Zero flag (did the operation result in a zero?). A Y appears in the column if the flag is affected by the instruction. An N indicates the flag is not changed by the instruction.*

## Sample Program 1

| Addr | Code | Mnemonic | Remark |
|---|---|---|---|
| 0 | %0001 | LDA #3 | ;Get first number |
| 1 | %0011 | | |
| 2 | %0100 | TAB | ;Move to B |
| 3 | %0001 | LDA #7 | ;Get second number |
| 4 | %0111 | | |
| 5 | %0000 | ADD | ;Figure sum |
| 6 | %1111 | JMP 6 | ;Jump self to stop |
| 7 | %0110 | | |
| 8 | %0000 | | |

## Sample Program 2

| Addr | Code | Mnemonic | Remark |
|---|---|---|---|
| 0 | %0010 | LDA 240 | ;Get first number |
| 1 | %0000 | | |
| 2 | %1111 | | |
| 3 | %0100 | TAB | ;Move to B |
| 4 | %0010 | LDA 241 | ;Get second number |
| 5 | %0001 | | |
| 6 | %1111 | | |
| 7 | %0000 | ADD | ;Figure sum |
| 8 | %0011 | STA 248 | ;Put low nibble in memory |
| 9 | %1000 | | |
| 10 | %1111 | | |
| 11 | %1110 | BCC 19 | ;Only one nibble answer |
| 12 | %0011 | | |
| 13 | %0001 | | |
| 14 | %0001 | LDA #1 | |
| 15 | %0001 | | |
| 16 | %1111 | JMP 21 | ;All done |
| 17 | %0101 | | |
| 18 | %0001 | | |
| 19 | %0001 | LDA #0 | ;Zero A |
| 20 | %0000 | | |
| 21 | %0011 | STA 249 | ;Put high nibble in memory |
| 22 | %1001 | | |
| 23 | %1111 | | |
| 24 | %1111 | JMP 24 | ;Jump self to terminate |
| 25 | %1000 | | |
| 26 | %0001 | | |

## Sample Program 3

| Addr | Code | Mnemonic |
|---|---|---|
| 0 | %0001 | LDA #2 |
| 1 | %0010 | |
| 2 | %0100 | TAB |
| 3 | %1000 | AND |
| 4 | %0110 | RRC |
| 5 | %0011 | STA 254 |
| 6 | %1110 | |
| 7 | %1111 | |
| 8 | %0000 | ADD |
| 9 | %0011 | STA 254 |
| 10 | %1110 | |
| 11 | %1111 | |
| 12 | %0000 | ADD |
| 13 | %0011 | STA 254 |
| 14 | %1110 | |
| 15 | %1111 | |
| 16 | %0001 | LDA #6 |
| 17 | %0110 | |
| 18 | %0011 | STA 254 |
| 19 | %1110 | |
| 20 | %1111 | |
| 21 | %0000 | ADD |
| 22 | %0011 | STA 254 |
| 23 | %1110 | |
| 24 | %1111 | |
| 25 | %0000 | ADD |
| 26 | %0011 | STA 254 |
| 27 | %1110 | |
| 28 | %1111 | |
| 29 | %0000 | ADD |
| 30 | %0011 | STA 254 |
| 31 | %1110 | |
| 32 | %1111 | |
| 33 | %0001 | LDA #13 |
| 34 | %1101 | |
| 35 | %0011 | STA 254 |
| 36 | %1110 | |
| 37 | %1111 | |
| 38 | %1111 | JMP 38 |
| 39 | %0110 | |
| 40 | %0010 | |



### Roundabout Addition

Sample Program 1 will add the numbers 7 and 3, and the answer will end up in the Accumulator. If you haven't already, turn on the power by pressing **P**. Now, press **B** for **B**egin, and confirm that the rotary is pointing at M (Memory). Now "key-in" this program with the following procedure:

1. Toggle the switches to the on and off positions corresponding to the bits of the number identified as Code in the program—up (or on) for 1, and down (or off) for 0. Notice that each binary code is preceded by a % (percent) sign to make it easy to distinguish binary numbers from decimal quantities (See "Numbers To Bits And Back" for details).

2. Check that the address indicated by the location counter is the correct one for that Code, and then Press **L** for Load.

3. Press **I** for Increment. This will take you to the next address.

4. Repeat steps 1 through 3, loading the correct nibble into each address, and move on to the next set until you've loaded all the nibbles in the proper order.

5. Once you have completed loading the program, press **B** again to return to address 0. Then step through each memory location with the **I** key to be certain the program is entered properly.

6. Now press **B** for **B**egin once more, then **R** for **R**un. Note that you may **H**alt the program at any time (by pressing **H**) and continue again by pressing **R**.

Let's go over Sample Program 1 step-by-step to see exactly what it does when **L**oaded and **R**un. First it uses the "LoaD Accumulator immediate" instruction (abbreviated LDA #) to load the number stored at the address immediately following the instruction code (address 1) into the Accumulator. This number (in this case a %0011 or decimal 3) is one of the two to be added. At address 2 is an instruction to Transfer the number from the Accumulator into register B (TAB). Address

*Photo 1: This shows the contents of the A register in the initial step of Sample Program 1. First, the program moves one number (3 or %0011) of an addition problem into A.*

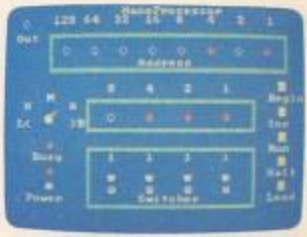


*Photo 2: Next, after the first number moves to the B register, the second number (7 or %0111) is loaded in A.*

*Photo 3. The A register now shows the result (10 or %1010) after the contents of A and B have been added together.*

3 contains another LDA# instruction to Load a %0111 (7 decimal) from address 4 into register A. The instruction at address 5 actually ADDs the number in register B to the number in A, and places the answer in A. Address 6 contains a JuMP instruction (JMP *addr*), that tells the machine to jump to the address specified at the next two memory locations—7 and 8. All addresses are two nibbles, and the *NanoProcessor* follows a procedure standard to many microprocessors where the low nibble of the address is in the next location (7 in this case) and the high nibble in the following one (8). We call this a "jump self" because we specify address 6 (%0000 0110) as the place to jump to.

When you **R**un this program, the "busy light" remains on and both rows of lights flash different patterns as the CPU steps through the program. The *Nanoprocessor* has been made to **R**un slowly so that you can track each instruction as it is executing. When the program "hangs-up" at location 6, press **H** (for **H**alt) to make the busy light go off. Now turn the rotary switch to point at A. Here you find the answer to the addition problem: %1010 or 10 decimal. Keep the pointer in this position and run the program again, after pressing **B**egin. Watch the A register change values—first 3 (%0011), then 7 (%0111), then the answer, 10 (%1010). Photos 1 through 3 show this sequence.

**CONTROL CAPSULE**
***NanoProcessor***

| Key | Function |
|---|---|
| B | Set address to zero. |
| I | Increment address by 1. |
| R | Run program. |
| H | Halt program. |
| L | Load location. |
| < | Move rotary switch counter-clockwise. |
| > | Move rotary switch clockwise. |
| P | Toggle Power switch. |
| E | End program (only when Power is off) |
| 1-4 | Toggle panel switch 1 = left-most bit, 4 = right-most bit. |

**CONTROL CAPSULE**
***NanoProcessor***

| Key | Function |
|---|---|
| OPTION | Save file. |
| SELECT | Load file. |

**CONTROL CAPSULE**
***NanoProcessor***

| Key | Function |
|---|---|
| F1 | Save file. |
| F3 | Load file. |

**CONTROL CAPSULE**
***NanoProcessor***

| Key | Function |
|---|---|
| CONTROL W | Save file. |
| CONTROL Q | Load file. |

**CONTROL CAPSULE**
***NanoProcessor***

| Key | Function |
|---|---|
| FN 6 | Save file. |
| FN 7 | Load file. |

**CONTROL CAPSULE**
***NanoProcessor***

| Key | Function |
|---|---|
| FCTN 6 | Save file. |
| FCTN 8 | Load file. |

## Moving On

In Sample Program 1, the machine added two numbers and got an answer that it could express in one 4-bit nibble. But, what if this answer had been larger than one 4-bit nibble—say, a number like 23 (%0001 0111)? Fifteen (%1111) is the largest number that one nibble can express. When a processor adds two numbers together whose answer is bigger than its registers can hold, the answer "overflows" the register. When this happens in *NanoProcessor*, a "carry flag" is set to 1 in a special Status register of the CPU. (This register is not directly accessible to the user.) The program has to contain commands that recognize the condition of this flag (either 1 when an overflow has occurred, or 0 when there is no overflow) and take appropriate action. You can determine which instructions cause changes in the carry flag by studying the C column (under "Flags affected") of Figure 3. If there is a Y in the C column, the instruction will affect the carry flag —i.e., set it to 1 if an overflow occurs, or reset it to 0 if no overflow occurs.

Sample Program 2 adds the numbers 11 (%1011) and 12 (%1100) to arrive at 23 (%0001 0111). Not only does the program have to check the carry flag, but because the answer doesn't fit in one register, it has to place the answer someplace else. The solution is to designate certain memory locations as data areas—two for input and two for output. Program 2 fetches the two numbers to be added from memory locations 240 (%1111 0000) and 241 (%1111 0001). These addresses are input areas. This means that before you **R**un the program, you must manually **L**oad the numbers to be added at these locations—place 11 at address 240, and 12 at address 241.

Similarly, the output area is at locations 248 (%1111 1000) and 249 (%1111 1001). The low nibble of the answer (%0111 in our example above) appears at 248, and the high nibble (%0001) at address 249.

This program also handles the overflow condition described above. If the answer does overflow a nibble, the program places a 1 in the accumulator and stores it as the answer's high nibble. If, however, the answer is less than 15 (and fits into one nibble), the program branches to another address, where it loads a 0 into A and stores that instead. This introduces one of 4 "conditional jump commands," which we will explore more fully in next issue's companion "utility," *NanoAssembler*.

Program 3 is a "mystery program" that actually accesses the "sound chip" we've built into the *NanoProcessor*. Watch next issue for an explanation of how this program works. Or perhaps, in the meantime, you will learn enough by playing with *NanoProcessor* to figure this one out yourself. The best way to learn the details of operating the the *NanoProcessor* is to use it and experiment by creating your own machine-language programs.

## Saving and Loading

With *NanoProcessor*, you can Save and Load the entire 256 memory locations (%0000 0000 through %1111 1111) to disk (and/or tape on The C-64, Atari, and TI-99/4A). Use the Save command listed in your Control Capsule and type in a file name in response to the prompt. To Load, use the Load command and type in the name of the file you wish to load.

**HCM Glossary terms:** CPU, bus, machine language, binary numbers, Random Access Memory (RAM), byte, address, nibble, location counter, accumulator, register, instruction set, mnemonic, branch, jump, conditional jump, status register, zero flag, carry flag, overflow, weight (of bits).

**HCM**

**For your key-in listings, see HCM PROGRAM LISTINGS Contents.**