THE No.1 MAGAZINE FOR ELECTRONICS TECHNOLOGY & COMPUTER PROJECTS

EVERYDAY PRACTICAL ELECTRONICS

Vol.34 No.10

CAN $6.99/US $4.95

USB THE EASY WAY

PIC-Based USB Interface

PHOTIC PHONE

A digital optical 'phone link

HALLOWEEN HOWLER

Uses digital WAV files in EEPROM

PLUS

BACK TO BASICS - 7

Parking Radar
Telephone Switch

www.epemag.co.uk

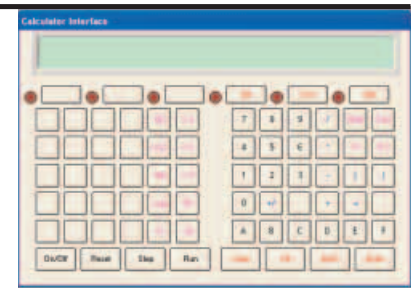
10

0 56698 56801 8

Printed in the UK

Special Feature

Introducing the Virtual DIY Calculator



Clive "Max" Maxfield & Alvin Brown

Learn how computers do maths without making your brain ache!

WHEN you come to think about it, there are lots of "application-type" computer books along the lines of *Learn Prof. Cuthbert Dribble's Visual Programming V6.0 In 21 Days* (you often have only 21 days, because that's when version 7.0 of the software is going to come out). Sad to relate, however, there really aren't many tomes – outside of mega-complex University courses – that teach how computers actually work.

In order to address this sad state of affairs, the authors decided to pen their own humble offering. One point we considered is that it's a lot easier to learn how to do something if you actually have a specific project in mind.

For example, if someone simply hands you a plank of wood, a saw, a hammer and some nails, you might hang around for a while pondering just what to do. But if you are also presented with the plans for a simple bird table, then you can immediately leap into the fray with gusto and abandon.

Thus, we decided to base a book (details are given later) on the concept of a simple calculator called the DIY Calculator, The cunning part of all of this is that we created the DIY Calculator as a virtual machine that runs on your home computer. This article is a spin-off from the book and is designed to give a brief introduction as to how the DIY Calculator functions. In order to facilitate this, you can download a fully-functional copy of the DIY Calculator software from our website at **www.DIYCalculator.com** for free (you'll find instructions on how to download and install the calculator on the website).

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), its memory devices (ROM and RAM) that are used to store programs (sequences of instructions) and data, and its 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. Read-only memory (ROM) has its contents hard-coded as part of its construction; by comparison, in the case of random access memory (RAM), you can load new values into it and read these values back out again later.

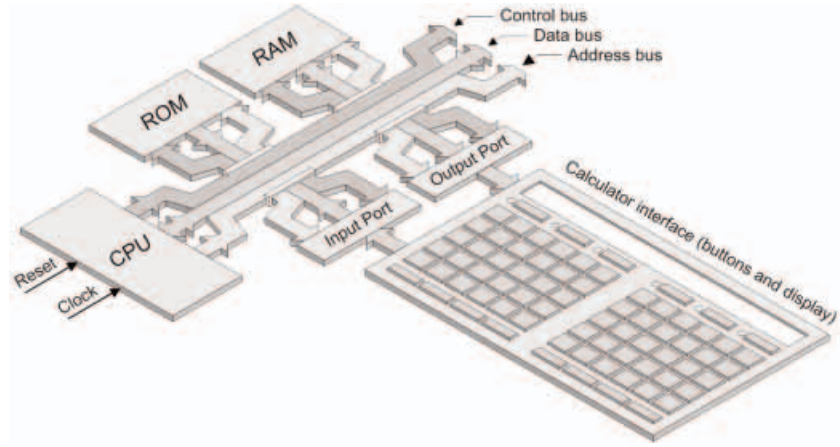


Fig.1. The main elements forming a computer-calculator.
Reproduced from How Computers Do Math *with the kind permission of the publisher*

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 eight 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, where the concepts of binary and hexadecimal are briefly introduced a little later in this article.

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 com-

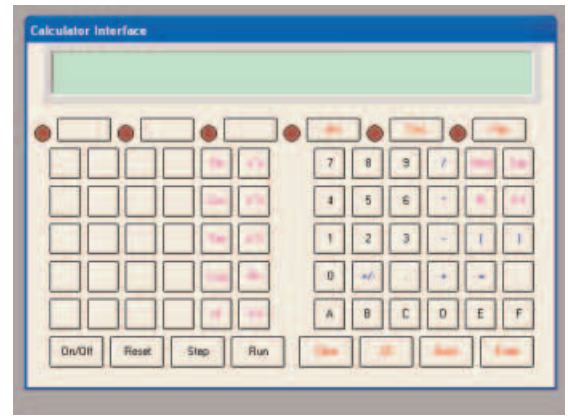


Fig.2. Screenshot of the DIY Calculator's interface

puters 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 (Fig.1).

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 rapscallion, which should look something like the screenshot shown in Fig.2.

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 computer programming 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 takes our source program and assembles (translates) it into 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).

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

Fig.3. Binary and hexadecimal

Listing 1: Program to read button codes and write to the main display

```
CLRCODE:    .EQU    $10          #   Code to clear main display
MAINDISP:   .EQU    $F031        #   Output port to main display
KEYPAD:     .EQU    $F011        #   Input port from keypad

            .ORG    $4000        #   Set origin to address $4000
            LDA     CLRCODE      #   Load ACC with clear code
            STA     [MAINDISP]   #   Store ACC to main display

LOOP:       LDA     [KEYPAD]     #   Load ACC from the keypad
            CMPA    $0F          #   Compare accumulator to $0F
            JC      [LOOP]       #   Jump if C flag is set
            STA     [MAINDISP]   #   . . . else copy ACC to display
            JMP     [LOOP]       #   Jump to LOOP

            .END                 #   End of the program
```

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.

Binary and Hexadecimal

In a moment we're going to create our own program, but before we start, we need to understand that computers store and manipulate data using the binary number system, which comprises just two digits: 0 and 1. One wire (or register bit/memory 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 data bus, this can be used to represent 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 Fig.3.

In this case, we use "$" characters to indicate hexadecimal values. Each hexadecimal digit directly maps onto four binary digits (and vice versa of course). This explains why we noted earlier that our 16-bit address bus could be used to point to 2^{16} = 65,536 different memory locations, which are numbered from $0000 to $FFFF in hexadecimal.

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 Fig.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.)

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 only concern ourselves with the carry (C) flag 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 (eight bits in the case of our virtual 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.

The Program Itself

For the purposes of this article, we're going to create a simple program that first clears the calculator's main display, and then loops around waiting for us to click one or buttons on the keypad. If any of these buttons are part of the "0" through "9" or "A" through "F" set, we're going to display these value on the main display. In a moment we're going to enter our program as shown in Listing 1. But before we do that, let's walk through this code step-by-step.

The first thing we do is to declare some constant labels and associated them with

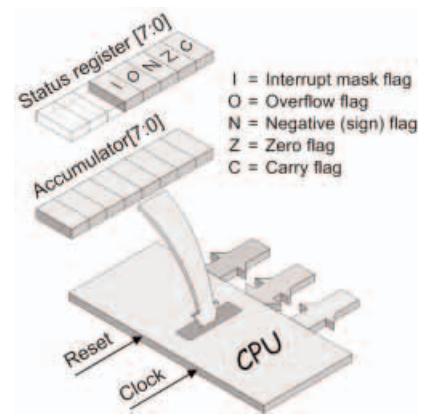


Fig.4. The accumulator (ACC) and status register (SR)

certain values using .**EQU** ("equate") commands. In the case of this program, the **CLRCODE** label is associated with a hexadecimal value of $10, which is a special code that will clear the calculator's main display. By comparison, the **MAINDISP** label is associated with the hexadecimal value $F031, which happens to be 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.)

Following the .**EQU** commands we see a .**ORG** ("origin") statement, which we use to specify $4000 as being the start address in our program. (The reason we use $4000 is that this is the first address in the DIY Calculator's virtual RAM. When we come to run the program, the DIY Calculator will automatically start at this address.)

Next, we use a **LDA** ("load accumulator") instruction to load our special clear code into the accumulator, and then we use a **STA** ("store accumulator") instruction to copy this value to the main display, thereby clearing it.

Now we find ourselves at the **LOOP** address label, which is where we are going to wait for the user to click on a key. There are a couple of things we need to understand here. First, our virtual calculator's front panel contains an 8-bit register. By default, this register is loaded with a dummy value of $FF. When we click a button on the keypad, a code associated with that button is loaded into this register. When the CPU reads from the input port connected to the calculator's keypad, it actually reads the value out of this register. Furthermore, the act of performing this read automatically reloads the register 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).

When we arrive at the **LOOP** label, we use a **LDA** ("load accumulator") instruction to load the accumulator from the memory location pointed to by the **KEYPAD** label. As we previously noted, this 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 with a value of $0F.

If the value in the accumulator is 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 "0" though "9" or "A" through "F", the **JC** ("jump if carry") instruction will return the program to the **LOOP** label to await another key. Otherwise, we use a **STA** ("store accumulator") instruction to copy this key code to the main display, and then we use a **JMP** ("unconditional jump") instruction to return us to the **LOOP** label to await another key.

The final statement in the program is a .**END** that, not surprisingly, informs the assembler that its task here is completed.

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 in Listing 1, use the assembler's **File > Save As** command to save this program with the name **mykeytest.asm**, and then use the assembler's **File > Assemble** command to translate your source program into a machine code equivalent called **mykeytest.ram** (if any errors are reported in the status bar at the bottom of the assembler window, debug them and reassemble 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 **mykeytest.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. The program is now looping around waiting for you to press a key. Try clicking several of the "0" to "9" and "A" to "F" keys and observe the corresponding characters appearing on the main display. Also try clicking some of the other keys (such as "=" and "+") and observe that – due to the way in which we created our program – these are discarded and do not appear on the display.

But Wait, There's More!

We've really 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. And if you want to experiment a little further, a slightly more testing example program that uses the DIY Calculator to implement a simple pseudo-random number generator is described on the website.

Further Projects

The book, *How Computers Do Math*, is organized in an interesting way. First there are a series of chapters introducing fundamental concepts such as *Subroutines* and *Recursion*. Each chapter is then backed up by a suite of interactive laboratories, each of which details what the reader will lean and how long it will take (typically 20 to 40 minutes each), 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, it is intended 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, 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.

The book, *How Computers Do Math* – John Wiley & Sons, ISBN 0471732788 – is scheduled to roll off the printing presses September 2005. Throughout *How Computers Do Math*, we introduce the way in which computers work and walk readers through the process of creating a simple four-function calculator program (add, subtract, multiply and divide) from the ground up. □