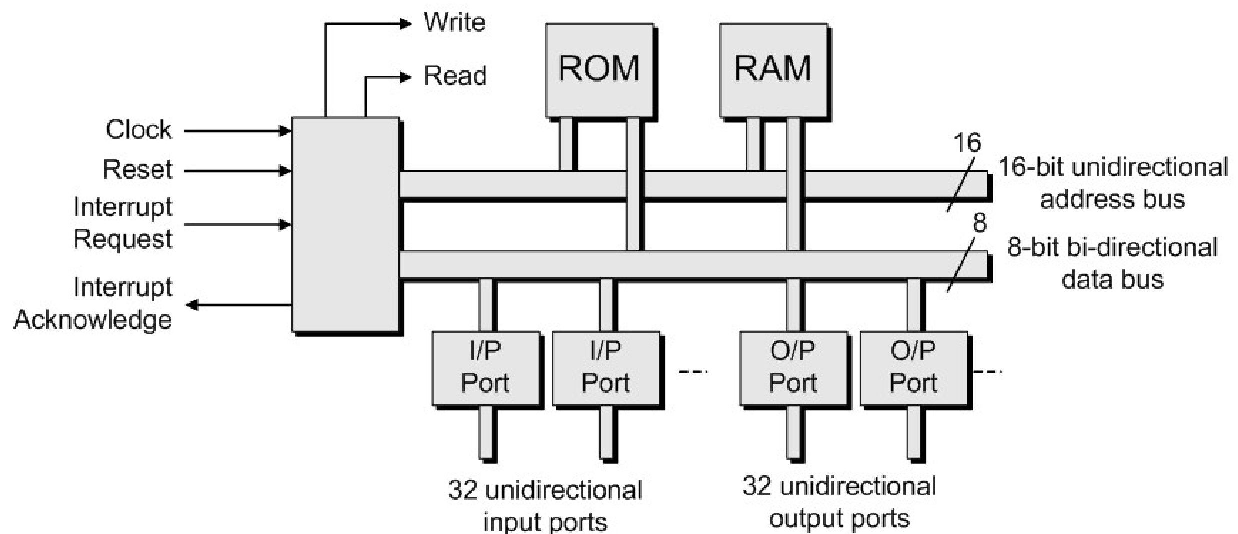
DIY Calculator: Physical Implementation

Introduction

The purpose of this paper is to describe some thoughts we've had with regards to creating a physical implementation of the DIY Calculator.

The Microcomputer Itself

As a brief overview, the virtual microcomputer powering the DIY calculator looks something like the following:



Note that the internal architecture of our (virtual) CPU – and also the architecture of the (virtual) microcomputer – is covered in detail in *The Official DIY Calculator Data Book* (this ~200 page book is provided for free on the CD accompanying our book *How Computers Do Math* (ISBN: ISBN: 0471732788).

Everything about this virtual system is designed to be as simple as possible. Hence the fact that we have separate read and write control signals, and we have unidirectional input and output ports. The output ports are actually 8-bit latches that continue to drive the last value written to that port. The actions/polarities of all of these signals are covered in detail in *The Official DIY Calculator Data Book.*

The 16-bit address bus potentially gives us 64KB of memory space. In our virtual world, this memory space is organized as follows:

- $0000 to $3FFF (the first 16K) is virtual ROM which we don't actually require in this incarnation of the calculator ('$' characters indicate hexadecimal values)
- $4000 to $EFFF (the next 44K) is virtual RAM
- Addresses $F000 through $F01F are occupied by a set of thirty-two input ports (of which the calculator uses only one port at address $F011), while addresses $F020 through $F03F are occupied by a set of thirty-two output ports (of which the calculator employs

Copyright © 2005 Clive "Max" Maxfield and Alvin Brown. All rights reserved.

only two ports at addresses $F031 and $F032). Finally, addresses $F040 through $FFFF are unused in this implementation.

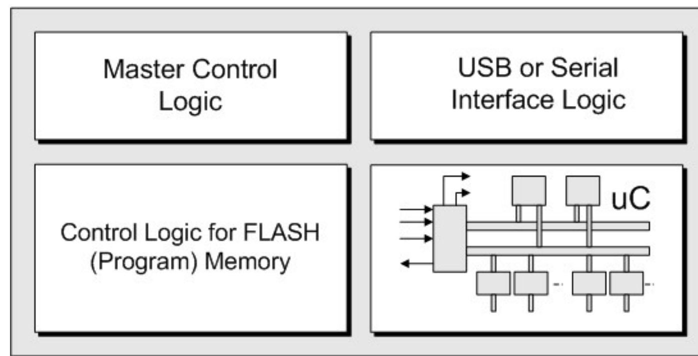
Not that, in our virtual world, all of our programs start at the first "RAM" location at address $4000. See also the discussions below on the way in which we might decide to implement different portions of this memory map in the physical world.

As was noted above, *The Official DIY Calculator Data Book* gives a detailed breakdown of the internal architecture of the CPU. This architecture – which is at the level of 8-bit and 16-bit register and multiplexers blocks and control signals etc -- was not designed to be efficient, but rather to be simple and understandable. *Appendix C* in the data book gives a complete clock-cycle-by-clock-cycle breakdown of the way in which each instruction type is executed.

Our idea is that the first physical implementation of the DIY Calculator (especially its CPU) should match the data book as closely as possible. Once the first implementation is up and running, it would be interesting to modify the architecture of the CPU so as to be more efficient.

An FPGA-based Implementation

There are a number of ways in which one might implement a physical version of the DIY calculator. However, the simplest approach would probably be to create a *Field-Programmable Gate Array (FPGA)*-based realization. Once programmed, the contents/functionality of the FPGA would be as follows:



The FPGA

The uC (microcomputer) block – shown to the lower-left of the above illustration – would implement the functionality of the DIY Calculator's CPU and also the rest of the DIY Calculator's microcomputer system (that's the CPU, ROM, RAM, and I/O ports). The idea is that (using one mechanism or another as discussed later in this document) we will load DIY Calculator machine-code files generated by our assembler into the RAM portion of the uC block and then execute these programs.

All of the DIY Calculator's control signals (read, write, clock, reset, interrupt request, and interrupt acknowledge) would be made available to the outside world via the FPGA's I/O pins. Input versions of these signals (e.g. reset and interrupt request) will have pull-up resistors holding them to their inactive states.

There won't be enough pins available on a low-cost FPGA to make all of the DIY Calculator's 32 input and 32 output ports available, so we suggest only implementing the ports that the

Copyright © 2005 Clive "Max" Maxfield and Alvin Brown. All rights reserved.

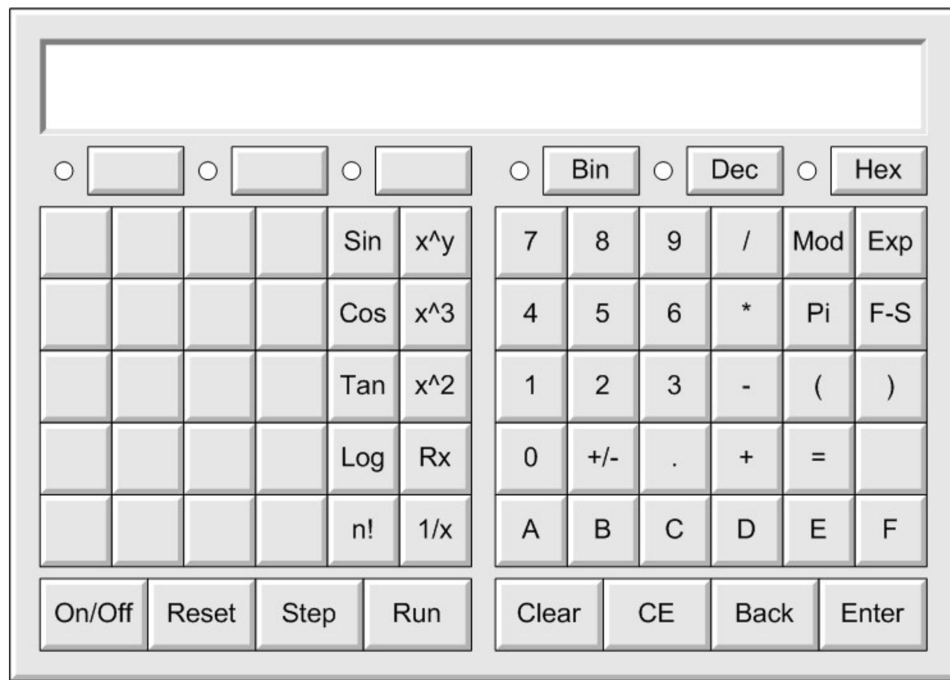
calculator actually uses. These would be the input port at address $F011 and the two output ports at addresses $F031 and $F032.

As was noted earlier, in our virtual world, the ROM (which is actually unused) occupies addresses $0000 through $3FFF, the RAM occupies addresses $4000 through $EFFF, and the I/O ports occupy addresses $F000 through $F03F. Furthermore, all of our programs start at the first "RAM" location at address $4000. If the FPGA is limited in memory (in the form of internal block RAM), then there is no need to implement the ROM in its entirety. All that would be required would be to implement three locations at addresses $0000 through $0002 and hard-wire these locations with an unconditional jump instruction to the start of the RAM at address $4000. Similarly, we don't need to implement all of the RAM (although it would be nice if we could), but instead we could implement as much as possible starting at address $4000. Last but not least, we could implement the three I/O ports used by the DIY Calculator individually (we don't need to implement all 64 ports).

The function of the other three blocks in the FPGA will be discussed later in this document.

The Calculator Board

As a reminder, the virtual DIY Calculator front panel looks something like the following:



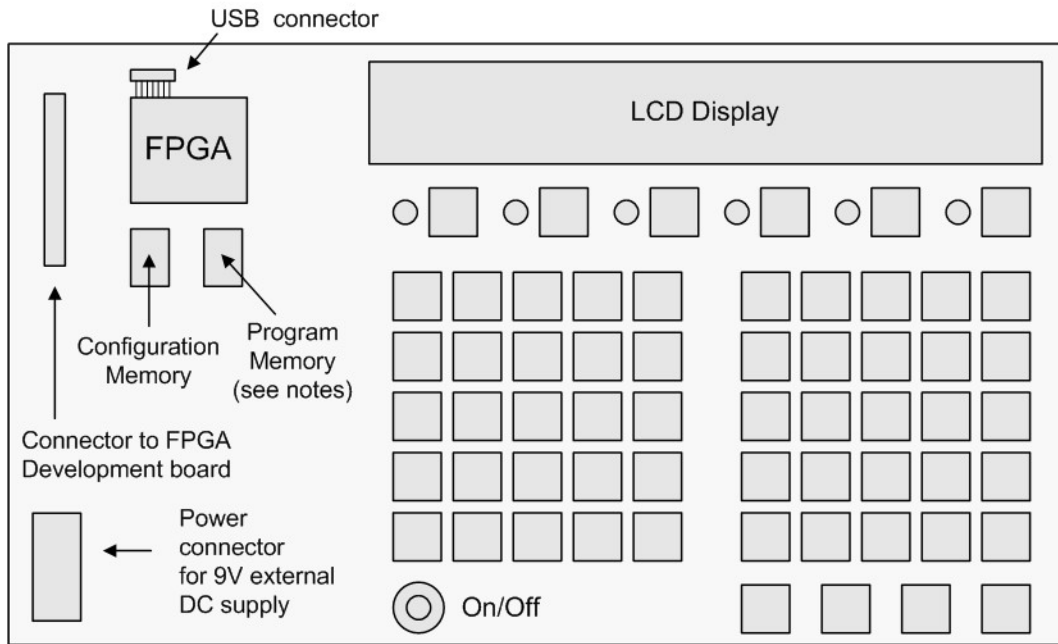
Observe that there are six LEDs – one associated with each of the buttons on the top row. Also observe the group of four buttons on the bottom left-hand side. The physical version could keep the ON/OFF button (perhaps as a physical toggle switch), but it would not require the other three, which are used in the virtual world for simulation control.

It would be nice if the physical version of the DIY Calculator used an LCD display, but the main display could be implemented in other ways such as using 7-segment displays (see also the discussions below). Similarly, it would be nice if the physical board sported all of the function

Copyright © 2005 Clive "Max" Maxfield and Alvin Brown. All rights reserved.

buttons provided in the virtual world, but it would also be possible to use smaller (cheaper) button arrays (see also the discussions below).

Moving on, the main Physical Calculator board could look something like the following:



As we see, this board would have the LCD (or other) display, six LEDs (although this isn't critical), the calculator buttons, an On/Off switch, and a power connector from an external 9V (or whatever) power supply.

Now there are two main scenarios here. One option would be for the FPGA controlling this board to actually reside on a separate FPGA development board. In this case, there would not be an FPGA as shown in the diagram above; nor would there be any configuration (FLASH) memory or a USB (or serial) connector. Instead, there would just be a main connector that would be used to link this board to the FPGA development board. Furthermore, we would not have the USB or Configuration Control Logic Blocks that were shown in the FPGA illustration earlier in these discussions.

Alternatively, the scenario that is predominantly illustrated above assumes that this physical calculator board is relatively self-contained (standalone), in which it will have an on-board FPGA, configuration (FLASH) memory, and – ideally – a USB or serial interface connector.

Usage (Reprogramming) Model(s)

Let's assume that we have opted for the second option discussed in the previous section; that is, to implement a free-standing calculator board. In this case, we have to give some thought to the usage model, especially with regards to providing the calculator with a program to run.

In our virtual world we have an assembler. Amongst other things, it generates machine code files for use in the virtual DIY Calculator. We call these RAM files. The format for these files is as follows:

Copyright © 2005 Clive "Max" Maxfield and Alvin Brown. All rights reserved.

```
# FILE TYPE: DIY Calculator RAM (*.ram) file
# GENERATED: DIY Calculator Assembler V2.0
# DATE-TIME: 29 May, 1957, 02:05 PM
# SOURCEWAS: C:/Program Files/DIY Calculator/Work/example.asm

STARTADDR: $4000
FINALADDR: $4142

$4000 01 90 80 43 28 65 78 95 28 65 78 95 01 90 80 43
$4010 43 28 65 78 95 28 65 78 95 01 90 80 43 80 43 28
$4020 78 95 28 65 78 80 43 28 65 78 28 65 78 95 80 47
$4030 01 90 80 43 28 65 78 95 28 65 78 95 01 90 80 43
$4040 43 28 65 78 95 28 65 78 95 01 90 80 43 80 43 28
$4050 78 95 28 65 78 80 43 28 65 78 28 65 78 95 80 47
$4060 01 90 80 43 28 65 78 95 28 65 78 95 01 90 80 43
$4070 43 28 65 78 95 28 65 78 95 01 90 80 43 80 43 28
$4080 78 95 28 65 78 80 43 28 65 78 28 65 78 95 80 47
$4090 01 90 80 43 28 65 78 95 28 65 78 95 01 90 80 43
$40A0 43 28 65 78 95 28 65 78 95 01 90 80 43 80 43 28
$40B0 78 95 28 65 78 80 43 28 65 78 28 65 78 95 80 47
$40C0 01 90 80 43 28 65 78 95 28 65 78 95 01 90 80 43
$40D0 43 28 65 78 95 28 65 78 95 01 90 80 43 80 43 28
$40E0 78 95 28 65 78 80 43 28 65 78 28 65 78 95 80 47
$40F0 01 90 80 43 28 65 78 95 28 65 78 95 01 90 80 43
$4100 43 28 65 78 95 28 65 78 95 01 90 80 43 80 43 28
$4110 78 95 28 65 78 80 43 28 65 78 28 65 78 95 80 47
$4120 01 90 80 43 28 65 78 95 28 65 78 95 01 90 80 43
$4130 43 28 65 78 95 28 65 78 95 01 90 80 43 80 43 28
$4140 78 95 28 -- -- -- -- -- -- -- -- -- -- -- -- --
```

Following a block of comments, we see two keywords STARTADDR and FINALADDR whose associated values (specified as hexadecimal numbers) reflect the start and end addresses of the program, respectively. With regard to the body of the RAM file, each line commences with a 2-byte hexadecimal address followed by sixteen 1-byte hexadecimal data values. In this context, the data values may contain opcodes, operands, or raw data.

So now let's assume that we have the physical calculator board. On power-up, the configuration FLASH memory on the FPGA board configures the FPGA to implement our virtual computer, the USB port, etc. Meanwhile, the second FLASH memory device would contain a DIY Calculator machine code program. The master control logic would read this program from the external FLASH memory and load it into the DIY Calculator's RAM (which would actually be realized as one or more block RAMs on the FPGA).

The point is that it would also be very useful if the user had the ability to download new programs into the calculator. This would require a small utility program to be created to run on the user's personal computer, where this program would read in a RAM file generated by the DIY Calculator's assembler and output it via a USB (or serial) port.

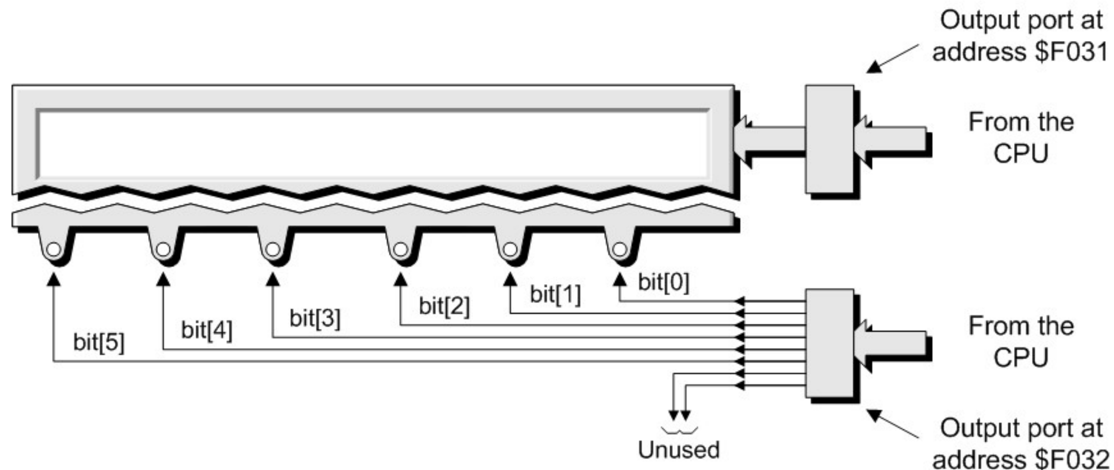
In this case, the master control logic in the FPGA would work in conjunction with the USB (or serial) interface logic and the control logic for the FLASH program memory. The FPGA would receive the new machine code program from the PC and program it into the FLASH memory

Copyright © 2005 Clive "Max" Maxfield and Alvin Brown. All rights reserved.

device, and also load it directly into the uC's RAM (that is, the appropriate FPGA block RAM). The next time the physical calculator is powered up, the master control logic would read the new program from the external FLASH memory and load it into the DIY Calculator's RAM (which, once again, would actually be realized as one or more block RAMs on the FPGA).

Driving the LEDS

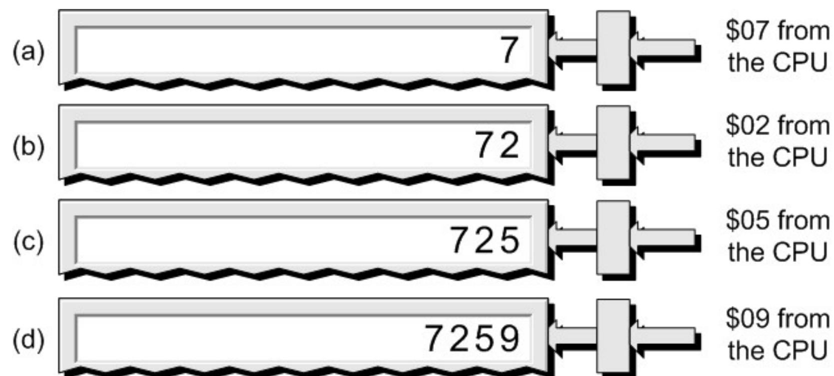
The way in which the LEDs are driven from Output Port $F032 is as follows:



The way this works in the virtual world is that a logic 1 turns the associated LED on while a logic 0 turns it off (this is the most intuitive way to think about things for beginners). If the LEDs are implemented on the physical board, then they should function in the same way.

Additional Calculator-specific Support Logic?

As a final consideration, it may be difficult to make some portions of the physical implementation exactly match the virtual one – so we need some way to bring them into sync. For example, consider the LCD display that is driven from the virtual computer's output port $F031 (see the image above). What isn't shown here is that there would also be some form of a write control signal. Furthermore, assuming this display to be initially cleared (more below), writing a series of codes such as $07, $02, $05, and $09 to this port will cause the display to show 7, 72, 725, and 7259, respectively.



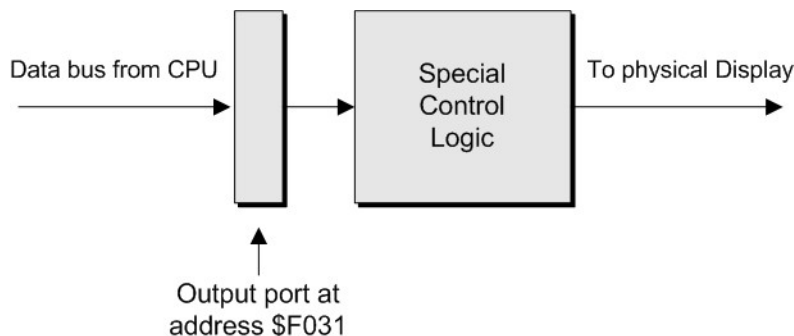
Copyright © 2005 Clive "Max" Maxfield and Alvin Brown. All rights reserved.

In fact, our virtual display understands the following codes:

Not Standard ASCII				Standard ASCII											
$00	0	$10	Clr	$20	SP	$30	0	$40	@	$50	P	$60	`	$70	p
$01	1	$11	Bell	$21	!	$31	1	$41	A	$51	Q	$61	a	$71	q
$02	2	$12	Back	$22	"	$32	2	$42	B	$52	R	$62	b	$72	r
$03	3	$13		$23	#	$33	3	$43	C	$53	S	$63	c	$73	s
$04	4	$14		$24	$	$34	4	$44	D	$54	T	$64	d	$74	t
$05	5	$15		$25	%	$35	5	$45	E	$55	U	$65	e	$75	u
$06	6	$16		$26	&	$36	6	$46	F	$56	V	$66	f	$76	v
$07	7	$17		$27	'	$37	7	$47	G	$57	W	$67	g	$77	w
$08	8	$18		$28	(	$38	8	$48	H	$58	X	$68	h	$78	x
$09	9	$19		$29	)	$39	9	$49	I	$59	Y	$69	i	$79	y
$0A	A	$1A		$2A	*	$3A	:	$4A	J	$5A	Z	$6A	j	$7A	z
$0B	B	$1B		$2B	+	$3B	;	$4B	K	$5B	[	$6B	k	$7B	{
$0C	C	$1C		$2C	,	$3C	<	$4C	L	$5C	\	$6C	l	$7C	|
$0D	D	$1D		$2D	-	$3D	=	$4D	M	$5D	]	$6D	m	$7D	}
$0E	E	$1E		$2E	.	$3E	>	$4E	N	$5E	^	$6E	n	$7E	~
$0F	F	$1F		$2F	/	$3F	?	$4F	O	$5F	_	$6F	o	$7F	

Note that the “Clr” code will clear the display, the “Back code” will remove the last character from the display, and all of the other characters will move one place to the right. The “Bell” code is supposed to make an annoying beep (but we would omit this from the physical implementation – still accept the code, just don’t do anything with it).

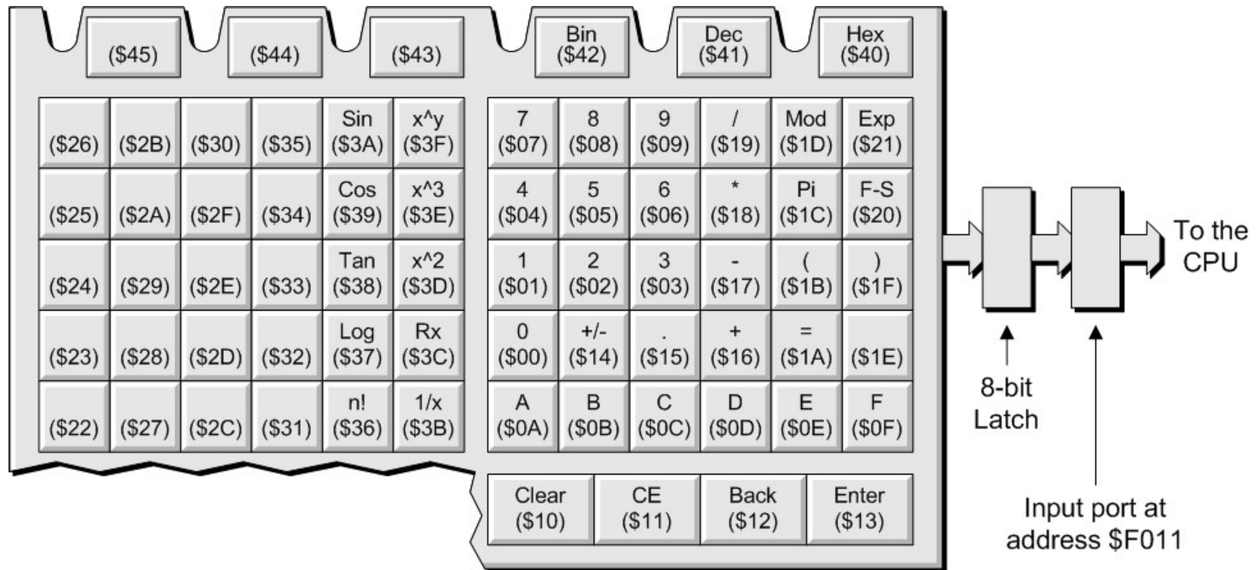
The problem is that the real physical LCD display will probably require a whole series of codes to initialize it. Also, it may well not understand our codes as shown above. One option would be to modify the DIY Calculator assembly program to write out the codes required by the physical display, but this is not a very attractive solution because it would mean that we would have two programs to maintain: one for the virtual world and one for the physical world. In fact, the preferred solution would be to add some additional logic into the FPGA as shown below:



Copyright © 2005 Clive “Max” Maxfield and Alvin Brown. All rights reserved.

When the calculator us first powered up, this special control logic block would automatically generate the initialization sequence required by the physical LCD (or other display). Subsequently, the special logic would translate any character (or control) codes coming out of the DIY Calculator's output port $F032 into equivalent codes that the real LCD understands. Thus, the fact that the physical display differs from the virtual display would be totally transparent to the user, and the same DIY Calculator programs would run in both the virtual and physical worlds.

Of course, there is another case that's calculator-specific, which is the way the buttons work on the calculator's keypad. Each button has an 8-bit code associated with it as shown below:



First, the virtual calculator has the concept of an 8-bit latch as shown above (this latch would need to be modeled/represented in the FPGA). When the calculator is first powered up, this latch is loaded (initialized) with a default $FF code. When a button is pressed, its 8-bit code gets loaded into the latch. If another button is pressed before the first code has been read out of the latch, then the code associated with the new button overwrites the old one. When the CPU reads from the input port at address $F011, the act of reading automatically reloads the latch with its default $FF value.

The point is that implementing these codes on the real keyboard may be tricky and expensive. There might be an easier/cheaper implementation solution. In such a case, we might wish to play the same trick as was discussed for the LCD display; that is, to add a special logic block that replicates the concept of the latch, eliminates any switch bounce, and translates button codes from the physical world into the codes we expect to see in the virtual world.

Copyright © 2005 Clive "Max" Maxfield and Alvin Brown. All rights reserved.