TANDY

The CoCo Column

by Dan Robins

I hope by this time, you've received at least a hint of the power of the Color Computer and the 6809 microprocessor chip. With this column, we finish up our series on the insides of the CoCo. Also, Kevin Darling details the problems with Tandy's and FD-502 disk drive. At this time, Tandy is aware of the problem, but it looks like the fix is up to you. Finally, a look at Spectrum Projects, one of the Color Computer's longest supports, and why the doors are now closed.

Back in the early days of computers, television monitors were an unheard of concept. Punch cards were used to enter programs and data into the computer. A line printer was about the only way to retrieve manipulated data. Now, personal computer companies compete for graphic's excellence. The question remains, how does it work?

With the CoCo, it is actually the work of three chips doing the main job of bringing the video information to the screen. It is beyond the scope of this article to detail how an RF modulator works. However, we can tell you how the information is fed to the modulator. Look at it though, as your own personal television station transmitter.

The four chips responsible for video transmission are the MC6847, MC6883, MC1372 and MC6821. MC6883 is the SAM or Synchronous Address Multiplexer chip. Fancy name for a clock chip, don't you think? Well, it is a bit more complex than that, but for video display it's VDG CLK signal is important. It produces a clock signal of 3.579545 megahertz, necessary to generate a color signal for TV. In technical terms, this frequency is necessary to meet the requirements for "scan lines" of a TV monitor. In addition to it's clock duties, it uses three registers (the Display Mode Registers), to set the proper resolution or type video output.

In a previous article, we explained how you could use the MC6821 PIA (Peripheral Interface Adapter) to set control registers in order to set the data lines as input or output. This same PIA allows us to use the control registers to help set the VDG chip into the proper display mode. These modes include two alphanumeric modes, five semigraphic modes, and eight modes of high resolution. Keep in mind though, a few of the semigraphic modes are not supported.

Select the mode that you wish by setting bits off or on (high or low) on V2, V1, V0 of the Display Mode Registers and all of the seven PIA register bits at memory location $FF22. Additionally, when poking the memory location with the desired character, the top two bits (bits 7 and 6) will implement the inverse video of the character and mode switching to any of the four semigraphic's modes.

What I would like you to imagine, is that you have a spray gun capable of three different colors. The graphics mode you are in determines the type of spray head you will use. The spray head will take care of the mixture of the colors and when they are to be applied. When spraying a color you only have so much time and a certain area to color. The time involved is a dot-clock, the number of clock ticks, and the area to cover is a scan line. And in this case, you'll be spraying your screen. But now don't do this in real life!

Using high resolution graphics, specifically the 256 by 192 PMODE4 graphic's mode, each element (or pixel) of the screen is painted either black or white (off or on). Each byte of information on the data line contains 8 pixels worth of information. Each pixel equals one dot-clock by one scan line. The 128 by 192 PMODE3 graphic's mode is different, as each pixel equals two dot-clocks by one scan line and the

continued on page 236

TEXAS INSTRUMENTS

TI Forum

by Jonathan Zittrain

Boston TI Faire A Great Success

The third annual Boston Computer Society TI Users Group fair was held Saturday, April 9. Attendance was estimated at around 300, a bit lower than past years, but sales seemed as brisk as ever. Everyone in attendance seemed to be having an excellent time, and the organization was near flawless.

I (JZ) was honored to be able to attend and meet quite a few TI luminaries. Mark Von Caponelles was working on offering a "Gram Kracker" alternative, the "Gramulator." RAVE '99 displayed their latest PC keyboards for the 99/4A, as well as a RAM disk. Genial Computerware unveiled the new PC-like database program FirstBase, and hopes to be shipping soon. Asgard Software had a booth filled to the brim with various software packages, new and old. CompuServe and GEnie had representatives Jim Horn and Scott Darling present.

There was a host of users groups from the northeast area, as well as flocks from Ottowa and Los Angeles. Myarc appeared close to releasing their new hard disk controller card, but was not quite yet ready.

During the day it was my pleasure to host a panel of 99'ers for an informal discussion of some of the issues confronting our community. The panel consisted of Jane LaFlamme, president of the Ottowa Users Group;. Barry Traver of Genial Computerware; Terrie Masters, LA 99'ers vice-president; Chris Bobbitt, the proprietor of Asgard Software; Lou Phillips of Myarc; Scott Coleman, West Penn. 99'ers president; and J. Peter Hoddie, BCS TI Group Co-Director and Coordinator of the fair.

Topics discussed included program piracy (and the users groups' possible contributions to it), program protection (Phillips discussed a new hardware-dependent system that Myarc is considering implementing for future software releases), and the nature of the TI market (can a software author expect to get a fair return on his or her investment of time and effort?). In future columns, starting with the next, I will be using the discussion as a springboard for the examination of many of these issues, after gaining more insights from different sources, including those that could not attend the fair. Future fair coordinators take note—getting a group of our community's leaders together for a discussion leads to interesting and beneficial results!

TELCO Clarifications

Charles Earl, author of the new TELCO program, has corrected two

continued on page 238

ATARI

Applying The Atari continued from page 232

work, remember?) in series to the jack and then to the existing speaker, you'll have an earphone jack that cuts off the internal speaker when the earphone is plugged in. Make sure you hook the wires to the right places. There are three tabs on the back of the jack. With the threads facing away from you, the left tab gets the common wire, the top tab gets the wire from the monitor to the speaker, and the right tab gets a wire from the jack to the existing speaker.

My Commodore 1702 monitor has the jack located behind the little front door and the left corner, but anywhere there is enough room for a quarter hold would be okay.

John Baum
Lisbon, Ohio

Newsletters

We'll pick more winning newsletters in next month's column. Meanwhile, here's a letter I received regarding the newsletter contest.

Dear Jeff:

While I appreciate greatly the fine column you write for *Computer Shopper*, the March '88 column made me bristle!

In the evaluation of the user group newsletters, you stated that "too many newsletters have pages overloaded with often inappropriate or meaningless Printshop and other icons." Your bias towards slick, magazine-like newsletters is apparent; such productions are well deserving of praise, since it is obvious the effort that goes into them is a labor of love. However, the Pinellas Atari Computer Enthusiasts' (PACE) newsletter is equally deserving of praise, produced completely on an 8-bit Atari engine (PACE is a group dedicated to the 8-bit Atari). While there may be laser printers that will tie to an 8-bit (although I doubt it), it is of interest to our group to keep the newsletter within the confines of the machines we deal with; as a result the "meaningless" Printshop icons have definite appeal.

Your selection of the "best newsletter" appears fully deserved; PACE would love to include advertising (there are no retailers in the area) and to include photographs of our activities in the newsletter—but this is overstepping the capabilities of our machines. Don't lose sight of two things: new owners still find "old" articles interesting (there are new 8-bit owners at each of our meetings), and the information contained in the newsletter may very well be "news" to the group, if the editor is aware of his readership.

Once again, I enjoy your column very much. But maybe instead of bashing the "poor" quality newsletters, give them credit for at least producing a banner that their organization finds worthy. I cannot speak highly enough of our newsletter editor, Thomas Davis, and his dedicated staff. Each issue of the P.A.C.E.SETTER gets better than the last, and I sincerely look forward to its delivery with anticipation—even without ads or pictures, and containing meaningless icons—because it has material that I find interesting. Thanks for your Atari support.

Alan Frazer, President
Pinellas Atari Computer Enthusiasts
Clearwater, Florida

Dear Alan:

I thank you for writing and hope I can clarify what qualities I look for when I select a winning newsletter. First of all, "meaningless Printshop icons" was not meant as a generalization of Printshop icons and I hope that others did not similarly misinterpret my statement. What I was referring to was the context in which specific icons are used. For example, I have just picked up a newsletter that has what appears to be a Mickey Mouse icon introducing a "Technical Notes" column. This is what I call a "meaningless" icon, in light of the fact that the image has absolutely nothing to do with the accompanying text. Upon browsing through your P.A.C.E.SETTER newsletter, I see a film projector with the "Next Meeting Preview" heading, a voltmeter icon for the "Local Repair" article, a pair of glasses with the "Library News" article. Now these are NOT meaningless icons, but are fine examples of how Printshop icons can be used.

Additionally, the amount of advertising in a newsletter is mentioned only as an indication of how successful the newsletter is in attracting advertisers, and is not used as a measure of the quality of the newsletter. For example, one recent newsletter winner had several full page advertisements from advertisers across the country. The advertisements did not make the newsletter a winner, but rather the reputation of being an exceptionally well-designed newsletter had attracted the advertisers.

Some of the best looking newsletters I have seen were produced entirely on the 8-bit Atari. Photos can always be directly incorporated into a newsletter to add more excitement, or they can be digitized via an 8-bit ComputerEyes. Any number of the dozens of 8-bit printing utilities can be used to further enhance the image of the newsletter. Don't feel that your newsletter is limited by an 8-bit machine; instead, seek out new ways for your 8-bit to churn out attention grabbing text and graphics.

Finally, I realize that many newsletters are produced on very tight budget or with very few people, and I would never intentionally "bash" any of them. Unfortunately, space does not permit me to list all of the newsletters in which groups gave it their best shot. Therefore, the "best newsletter" title goes to the newsletter that combines a wide range of non-reprinted articles, useful graphics and illustrations, excellent organization, etc.

Next Month

More reader mail and requests, news and reviews for your 8-bit Atari.

Address all correspondence to: Jeff Brenner, "Applying The Atari 7/88," c/o *Computer Shopper*, PO Box F, Titusville, FL 32781. ●

More Atari continued on page 241

TEXAS INSTRUMENTS

TI Forum continued from page 238

BASIC is criticized because it permits "spaghetti code" and doesn't force one to do structured programming. But isn't this criticism really an admission of the flexibility of BASIC? One can write very structured programs in BASIC, but a person can also write a "quick and dirty" program as well if he so chooses. It's up to the programmer, and there are times when a "quick and dirty" program is what's needed.

Similarly, BASIC is criticized because of its inclusion of a GOTO statement. It is interesting, however, that assembly language (the language closest to being the native language of the machine) has its own form of GOTO, because that is what B (branch) or JMP (jump) really are. And if GOTOs are so bad, why do people often spend so much time thinking of ways to add them to languages in which they are not innate?

Well, BASIC means Beginner's All-purpose Symbolic Instruction Code, which supposedly proves that it's only for amateur programmers, or so another argument against BASIC commonly runs. (Symbolic Instruction Code, by the way, merely means Language, but BAL wouldn't have made as clever an acronym.) But the fact that BASIC is accessible by more programmers than any other language seems to me to be a virtue rather than a defect. And BASIC is an "All-purpose" language, which declares that it is more versatile than many other languages, e.g., Cobol, Fortran, etc.

If an ordinary TI'er is running an assembly language program and wants to change it, ordinarily he's out of luck. But if the program is written in BASIC (TI BASIC or, more likely, TI Extended BASIC), then he usually can customize it easily in any way he wishes. This, I would think, would seem to be an important factor for those seeking to survive in an orphan community, with little significant support from major commercial software houses.

There are, in my mind, only one legitimate argument against BASIC: the argument that there are certain situations where BASIC does not provide sufficient speed or power to perform certain necessary tasks. Most of the time the commands of BASIC are entirely adequate, especially in interactive-style programs. But what about those situations where BASIC or Extended BASIC does not have the speed or power which is required?

The answer is to extend BASIC further, by adding new commands or subroutines. These new commands may be placed in the module (these are then usually accessible even to those TI'ers with minimal systems) or they may be assembly language subroutines (which need a memory expansion of 32K in which to reside). (Another theoretical answer to the matter of speed would be a suitable BASIC compiler, but in my opinion we do not yet have a thoroughly satisfactory one for the TI-99/4A. Besides, a BASIC compiler may provide an answer for increasing the speed of BASIC, but it does nothing to increase the power of BASIC.)

Just as TI Extended BASIC is actually easier to program in than TI BASIC because of the additional commands, so also "Extended Extended BASICs" are easier to use than "regular" Extended BASIC. The additional commands may include such things as allowing you 40 columns on the screen (instead of the usual 28 or 32), accessing other special graphics modes (e.g., bitmap), peeking and poking in VDP RAM (rather than merely CPU), direct sector access of disks (if you have a disk drive), manipulation of strings and arrays, and much more.

Next month's article will survey some of the specific "Extended Extended BASICs" that are available for the TI-99/4A. Some are commercial software (like Jim Hollender's SXB or Super Extended BASIC, and Chuck Davis' DEP or Display Enhancement Package), while others are fairware programs (like Michael Riccio's STAR or Super TI Assembly Routines, and Curtis Alan Provance's EDP or Enhanced Display Package). Some need a special module (like Mechatronics' Extended BASIC II +, or Triton's SEB or Super Extended BASIC), some are disk-based (like Richard Mitchell's String Master, or my own XXB or Extended Extended BASIC), and one requires a special module, is disk-based, and requires a special RAMdisk (Myarc's Extended BASIC II). Except for the last, none of these costs more than $50 or $60, and most cost much less, but all of them dramatically extend the power of what TI Extended BASIC can do.

Incidentally, I assume that many of the readers of this article do have a 32K memory expansion for their computer, but if you do not, then you should be aware that it is possible to add it inexpensively (perhaps $20 in parts, I'm told) by inserting it directly into the console. (Plans are available on how to do such a "matchbox conversion" for those who are interested.) You do not have to have a disk system to make use of most of the resources mentioned in the previous paragraph, but many of them do require the 32K memory expansion.

In line with the theme of "Extending Extended BASIC," this piece is being extended into next month's issue. ●

More Texas Instruments continued on page 390

Tour de Forth

by Glenn Davis

To work effectively with our computers—very complex machines—we must simplify our ideas about them. Such a simplification is called a "model." Chemists use tiny balls connected by springs to model atoms and molecules. Physicists make approximations, which are a type of model. Economists "model" the economy. You may have built model cars at one time. We use them so we can ignore detail.

The kind of "model" I'm concerned with here is the "programmer's model." Every computer has one or more programmer's models—how the programmer views the machine—or can view the machine. Many beginning Assembly or Forth programmers do not have the necessary Computer Science background to understand why things work the way they do. Understanding the way semiconductor memory is organized is usually a sore spot.

What Is In Memory?

Very often, people write out the contents of memory as a binary number such as 0011 1000 0100 0000. Each bit in memory can be on (1) or off (0). A combination of such binary digits (hence, bit) is a string or pattern. A string of bits can be any length we have memory to store: 2, 4, 8, 16 and 32 bits are common for microcomputers. Now, strictly speaking, strings are only a model for the contents of memory. We do not know if any given bit of a bit-string has any physical connection to the other bits in a memory-word, except that the computer addresses them as a unit. The bits usually have some physical connection, but that needn't concern us.

The smallest addressable unit for the 9900 series is the byte (a string of 8 bits). The fundamental unit is the "word" (16 bits for this processor, other processors have different word sizes, which is how they become known as 8-, 16-, or 32-bit processors). Note: the use of "word" differs between Assembly language and Forth. In Assembly it is a unit of memory. A "word" is an executable command in Forth. "Cell" describes this memory unit in Forth.

When we speak of the "least significant" bit of a string of bits, we have already modeled memory to indicate "place value." That is, the bits are ordered. Since people write the most significant digits on the left, and computers don't care, we generally write binary numbers in the same way: most signifcant bits on the left, least significant on the right. A 16-bit quantity can represent 65536 distinct states. To interpret these states, a convention is decided upon. In our case, and that for most microprocessors, we use "two's-complement" notation which specifies bit-patterns for both positive and negative numbers. There are other notations (one's-complement, signed magnitude, and radix-n among them) but they have fallen into disfavor. However, a given bit pattern could mean a number of things (within certain limits), depending on the use the programmer puts it to.

The pattern in the example above is >3840 in Hexadecimal notation. In the TI world, ">" indicates Hex notation; the rest of the world uses a "$" so $3840 is the same number. Once again, Hex is just a model for the bit pattern that actually exists in memory. I could just as easily have given it in Octal (base 8) as 034100. The numeric base is important in interpreting bits, especially since Forth can interpret all number bases.

This particular bit pattern would be the machine instruction MPY R0,R1 if the CPU tried to execute it. This is nothing more than a bit pattern that the designer decided would cause an unsigned multiplication. There is nothing instrinsic about >3840 that makes it a CPU instruction. To another CPU, >3840 might be nonsense, a MOVe instruction, or something else entirely. We can put such patterns into memory with an assembler, debugger, loader, or program file.

However, since the word >3840 can be modeled as two 8-bit strings (bytes) it is also the ASCII codes for the characters "8" and "@." If you saw >3840 in memory, you won't know what it is for unless you "why" that pattern is there. If the adjacent memory locations contain other ASCII characters, it is likely that these are; if they contain recognizable machine code, then it is probably the MPY instruction given earlier.

As with every other rule in the Universe, there are exceptions. Finding the memory words >0200 >3840 does not mean they are both machine instructions, since >0200 is obviously not printable ASCII text. >0200 is a machine instruction. >3840 in this case is a data value (which is 14,400 in deci-

continued on page 395

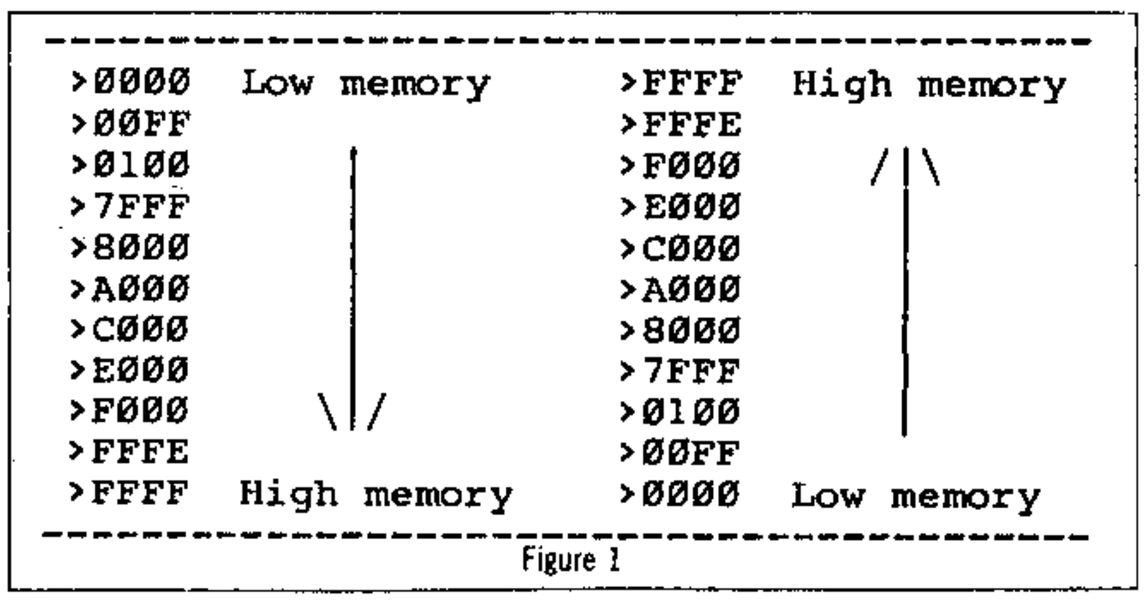

Figure 1

	TI/Motorola		Intel/IBM	
Address	word contents	even byte	odd byte	even byte	odd byte
>0000 :	>2697	>26	>97	>97	>26
>0002 :	>FF10	>FF	>10	>10	>FF
>0004 :	>2211	>22	>11	>11	>22
>0006 :	>1234	>12	>34	>34	>12

Figure 2

Tour de Forth
continued from page 390

mal). The assembler format is LI R0.>3840, that is, load the register R0 with the binary pattern >3840. Perhaps R0 will be used as a loop counter.

Some instructions—such as MOVe and Add—require one or more words of memory following them as addresses. MOV @>3840,@>8300 requires three words of memory: >C820 >3840 >8300. The middle word, our example from before, in an address of a memory location this time.

Memory bits are usually assigned place-values in addition to the memory address they reside in. If you think of memory addresses like residential street addresses (this is another model), bit numbers are like apartment numbers. Once again, TI did things backwards from the rest of the world by numbering the Most Significant Bit "0" and the Least "15" (for a 16-bit value.) Figure 3 uses this labeling. On the other hand, the rest of the world numbers the most significant bit as "15" and the least as "0." So when you are reading manuals or see references to "bit 2," remember the direction in which they are numbered.

On the other hand, bits don't have to be ordered. They could be Booleans (BOO-lee-an) which signify on/off, true/false or any combination of two or more states. As you will see, the values in a "Ts" field of a 9900 instruction have no intrinsic meaning or numeric value. It is a string of two bits that TI gave meaning to.

Memory Organization

This brings me to an important point: just as we model the contents of memory, we can model the way the computer addresses this memory. When working with ASCII text or other byte-size data, you may consider memory to be organized as on an elementary school number line. Zero in the middle, negative to the left, positive to the right. If we were to store the characters string "Texas Instruments" beginning at location zero, "T" would be followed by "e" in location 1, "x" in 2, "a" in 3, and so on until we put "s" in location 16. The space between "s" and "I" would also get its own byte. The Assembler's TEXT directive does this for you.

16-bit or word size data needs a different model. Two are commonly used, both are vertical. The first, and my personal favorite, is a "word-wide" model with memory location zero at the top, running downward. Successive words are later. The Assembler listing uses a similar format, and it is convenient since English is read left-to-right top-to-bottom.

The other vertical model runs in the opposite direction. Most computers using Intel processors use this model. Zero begins at the bottom (as though measuring something from the floor) and higher addresses run upward. That is, "low" memory is low on the page, "high" memory is high on the page. Leo Brodie's Starting Forth uses this type of memory map. See Figure 1 page 390.

A variation on these models is based on the byte order. The 9900 stores 16-bit values in two consecutive bytes, so memory words are stored at even addresses, beginning with zero so the first word of memory consists of the two bytes at >0000 and >0001. It stores the Most Significant Byte of a multi-byte value in the lower address (the even one) and the Least Significant Byte at the higher address (the odd one). This is also true for the M68000. Some processors (6502, Z80, 8088) store the MSB and LSB in the opposite order and/or do not require a 16-bit word to begin on an even address. With these CPUs, the LSB is at the lower address. Very technical reasons exist for these differences, and some programming models are processor-dependent for this reason. Forth was originally written to be byte-order independent, so only careless programming creates problems. See Figure 2 on page 390.

One more note on memory organization: strings are stored with the first character in the lowest memory address. The remaining characters in the string are stored at increasing addresses, as in the example mentioned earlier. This is true for both the TI/Motorola and Intel/IBM organizations.

A Step Backwards?

Given that the Geneve has the 9995 CPU instead of the 9900 CPU that the 99/4A has, what can we say about this

continued on page 465

Tour de Forth
continued from page 395

processor? The Atari ST line was so named because its processor is the "Sixteen/Thirty-two" bit Motorola 68000. The M68000 has 16 32-bit registers, but only a 16-bit data bus. The 9995 processor, with 16 16-bit registers and an 8-bit data bus, is an "8/16-bit" processor by this definition, unlike the 9900 which has a 16-bit data bus.

The 9900 has sixteen interrupt levels (0-15), whereas the 9995 has only five (0-4). The 4A only used interrupts 0, 1, and 2 anyway. Additionally, the 9995 has four more instructions, as well as an "overflow interrupt" which allows arithmetic calculations to trap overflows (such as dividing by zero) instead of explicity testing for them, saving considerable execution time in some circumstances. To be active, the Overflow Enable bit in the Status Register must be set.

A Giant Step Forward?

On the other hand, the 9995 in the Geneve runs at 12 MHz. For the IBM world, this is quite fast—the original IBM PC run at 4.77 MHz while many turbo machines run at 8 or 10 MHz. The Atari ST, Amiga, and Macintosh run 7-8 MHz. The catch is that the 9995 does an internal divide by four, so it is actually running at 3 MHz—the 99/4A's clock speed and about the same as the Apple IIgs.

However, each 9995 instruction takes fewer cycles to execute than the 9900, so throughput is 3-4 times better (depending on the mix of executed instructions). It is nearly impossible to compare raw clock speeds of processors from different families.

Does any of this really make a difference? No. The 4A's 9900 was strapped down by having only 8-bit wide memory, so two 8-bit bytes had to be fetched from memory and presented to the processor as 16-bits. The 9995 also has "pre-fetch" and "post-store" so memory speed is much less of a problem.

These are the new instructions that are implemented on the TMS9995: LST (Load STatus register from workspace register), LWP (Load Workspace Pointer from workspace register), MPYS (16 * 16 => 32 bit MultiPlY, Signed), and DIVS (32 by 16 bit DIVision, Signed). Adding these instructions to the Forth assembler is quite easy (see the Forth screens). The E/A assembler CAN work with the op-codes, but not with the mnemonics for the op-codes.

But first, here are the definitions of the instructions:

continued on page 466

```
SCR #54
  0 ( CONVERT TO GRAPHICS2 MODE CONFIG 14SEP82 LAO)
  1 0 CLOAD GRAPHICS2 BASE->R DECIMAL 56 R->BASE CLOAD SETVDP2
  2 BASE->R HEX : GRAPHICS2  0A0 1 VWTR
  3 -1 1B00 1800 DO 1+ DUP 0FF AND I VSBW LOOP DROP
  4 1 PABS @ VSBW 16 PABS @ 1+ VSBW 1 ( #FILE) 834C C! PABS @ 8356 !
  5  0A 0E SYSTEM ( SUBROUTINE TYPE DSRLNK TO SET 2 DISK BUFFERS )
  6 0 1800 0F0 VFILL ( INIT COLOR TABLE )
  7 2000 1800 0 VFILL ( INIT BIT MAP )
  8 20 SCRN_WIDTH ! 1800 SCRN_START ! 1B00 SCRN_END ! 1B00 PABS !
  9 1C00 DISK_BUF ! ( USER VARIABLES NOW SET UP )
 10  2 0 VWTR      6 2 VWTR  ( SET VDP REGISTERS )
 11 07F 3 VWTR     07 4 VWTR
 12 70 5 VWTR      7 6 VWTR
 13 0F1 7 VWTR     0E0 DUP 83D4 C! 1 VWTR    1BC0 836E ! ( VSPTR )
 14 0 0 GOTOXY 4 VDPMDE !  0 837A C!  ;
 15 R->BASE
SCR #75
  0 ( ASSEMBLER 12JUL82 LCT)  0 CLOAD A$$M
  1 BASE->R DECIMAL 74 R->BASE CLOAD ;CODE
  2 BASE->R HEX
  3 ASSEMBLER DEFINITIONS
  4 : GOP' OVER DUP 1F > SWAP 30 < AND
  5        IF + , , ELSE + , ENDIF ;
  6 : GOP <BUILDS , DOES> @ GOP' ;  ( Changes for 9995 20Dec87 GED)
  7 0440 GOP B,     0680 GOP BL,     0400 GOP BLWP,  01C0 GOP MPYS,
  8 04C0 GOP CLR,   0700 GOP SETO,   0540 GOP INV,   0180 GOP DIVS,
  9 0500 GOP NEG,   0740 GOP ABS,    06C0 GOP SWPB,
 10 0580 GOP INC,   05C0 GOP INCT,   0600 GOP DEC,
 11 0640 GOP DECT,  0480 GOP X,
 12 : GROP <BUILDS , DOES> @ SWAP 40 * + GOP' ;
 13 2000 GROP COC,  2400 GROP CZC,  2800 GROP XOR,
 14 3800 GROP MPY,  3C00 GROP DIV,  2C00 GROP XOP,
 15 -->
SCR #77
  0 ( ASSEMBLER 12JUL82 LCT)
  1
  2 : ROP <BUILDS , DOES> @ + , ;
  3                              ( Changes for 9995 20Dec87 GED)
  4 02C0 ROP STST,  02A0 ROP STWP,  0080 ROP LST,  0090 ROP LWP,
  5
  6 : IOP <BUILDS , DOES> @ , , ;
  7
  8 02E0 IOP LWPI,  0300 IOP LIMI,
  9
 10 : RIOP <BUILDS , DOES> @ ROT + , , ;
 11
 12 0220 RIOP AI,   0240 RIOP ANDI,
 13 0280 RIOP CI,   0200 RIOP LI,
 14 0260 RIOP ORI,
 15 -->
```

```
Status Register in the 9995:

 0  1  2  3  4  5  6  7  8  9  10 11 12  -  15
 ------------------------------------------------
|L>|A>|= |C |OV|OP|X |nu|nu|nu|OE|nu|I0|I1|I2|I3|
 ------------------------------------------------
Note: OE is Overflow Enable.

Format   Type   Status bits    Opcode      Effect
-------------------------------------------------------------
MPYS G     6       *0 - 2       01C0       MSW((R0)*(G))->(R0)
                                           LSW((R0)*(G))->(R1)
DIVS G     6       *0 - 2, 4    0180       INT((R0,R1)/(G))->(R0)
                                           REM((R0,R1)/(G))->(R1)
LST R      8        0 - 15      0080       (R) -> ST
LWP R      8        none        0090       (R) -> WP

Note: R is an arbitrary workspace register
      G is a general address
      * result compared to zero and status bits set
        accordingly

Instruction format 6:
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
 ------------------------------------------------
|            Opcode             | Ts |  Source  |
 ------------------------------------------------
Ts Address Mode:
Code  Mode                  Example
-------------------------------------------------
 00   Register Direct       Rn
 01   Register Indirect     *Rn
 10   Symbolic (S = 0)      @Label
 10   Indexed (S <> 0)      @Label(Rn)
 11   Indirect, auto-inc    *Rn+
Instruction format 8:
 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
 ------------------------------------------------
|            Opcode             |nu|  Register  |
 ------------------------------------------------
Note: nu indicates bit is 'not used'
```

Figure 3

Tour de Forth
continued from page 465

MPYS takes only one argument. This is one of the two operands for the signed multiplication. The other operand is implied, and is always R0. The 32-bit signed result is then stored in the register pair R0,R1—the most significant 16-bits being in R0. See Figure 3 on page 465 for specifics.

DIVS also takes just one argument, which is the divisor. The other operand is the implied register pair R0,R1 containing the signed 32-bit numerator. Following the division, the signed 16-bit dividend is stored in R0, while the 16-bit signed remainder is stored in R1. This is the "rounded" division that you learned in school, not the "floored" division that Forth-83 uses (see March 1988 *Computer Shopper*). If the numerator is only 16 bits, the high bit of R1 must be copied into each bit of R0 (simple sign extension) for the result to be correct.

LST simply loads the Status Register from a workspace register, where the interrupt mask and all of the status bits lie. The most obvious use is to set the Overflow Enable bit that I mentioned earlier. This is the opposite of the STST (Store Status Register) instruction.

LWP loads the Workspace Pointer from a workspace register. Formerly, only LWPI and BLWP could change the WP. This instruction makes stacking of workspaces much easier, as well as allowing software to change workspaces without a context switch. STWP (Store Workspace Pointer) is the opposite of LWP.

But, How To Assemble Them?

To use these new op-codes, you have to have some way of assembling them. Fortunately, the TI Forth 9900 Assembler included on the TI Forth disk is easily modified to accept the new op-codes. The screens to be modified are below. The op-codes MPYS, DIVS, LST, and LWP are all 9995 instructions for the Geneve. Do NOT try to use them with a 99/4A.

The screens are numbered as they are on the TI Forth system disk. Make the necessary changes to the screens by examining the differences (they are under the new comments on the right side of each screen), FORGET whatever you need to (type ASSEMBLER FORGET GOP' to remove the assembler from memory), recompile (type —ASSEMBLER), and perhaps BSAVE it (perhaps as I demonstrated in the January 1988 *Computer Shopper*).

The 99/4A Editor/Assembler cannot be modified so easily. Unless you did some serious patching, good ol' reliable E/A won't recognize the new mnemonics. However, that won't stop hardy souls from making the E/A assemble the op-codes anyway. This is the whole point in discussing "models." Even though the mnemonics aren't recognized, the CPU will properly execute the instruction if we present it with the proper bit-pattern.

What you can do with the E/A is use the DATA directive to initialize the contents of memory. When the CPU's Program Counter gets to the instruction it will treat it like any other instruction. See Figure 3 for data that will allow you to use the new op-codes. Here is an example:

```
MOV @NUMBER,R0
MOV @NUMBER + 2,R1
MOV @DIVISR,R2
DATA >0182
MOV R0,@INT
MOV R1,@REM
```

This code fragment does a signed division of the 32-bit quantity stored at NUMBER and divides it by the signed 16-bit quantity in DIVISR. After the division, the integer part is then moved to INT and the remainder is moved to REM. >0182 stands in for the "missing" mnemonic DIVS R2. As Figure 3 shows, >0180 is the basic op-code. To use R2 as the divisor, a "2" is added to the op-code since the Ts field is binary 00 and "source" then specifies the register number. If R2 indirect (*R2) is to be used, binary 01-0010 is added (>0180 + >0012 = >0192). Very simple. Other Ts values are in Figure 3. For MPYS and DIVS, remember that address modes besides register direct and register indirect require an additional address which must be provided by you: e.g. DATA >01A0,DIVISR is equivalent to the mnemonic DIVS @DIVISR and could be used in place of two lines in the code above.

The 32-bit quantity must always be in the register pair R0,R1. The other op-codes are assembled similarly. Be sure to include a comment on lines that use these op-codes so you remember their purpose. Also, test a small portion of code before creating the World's Greatest Program. If/when the rumored 9995 Macro Assembler is released, I recommend that you change the source from this to actual mnemonics.

Crush That Bug!

TI Forth has a tiny bug that appears when run on the Geneve. The 64-column editor did not work. You'd just get a mostly-blank screen instead of seeing Forth source when you tried to edit it. Fortunately, the bug is really easy to fix. The only change that needs to be made is in the code that controls the base address for the pattern table. In TI Forth this table is at >2000 in VDP RAM. Since the TI had only 16K VDP RAM, using table addresses that were too high had no damaging effect. This is not true for the Geneve (it has 128K VDP RAM), so we need to move the table back down where it is supposed to be. So, in line 11 of screen 54, change "0FF 4 VWTR" to "07 4 VWTR." See SCR#54 for the example. That is the only change that needs to be made. Be sure to reload the GRAPHICS2 code, especially if you use a BSAVEd Forth.

This is it for this month. Soon I'll provide some more modifications to the Forth Assembler, which will be useful to TI 99/4A and Geneve users. ●