ORION Instruments

Applications Notes
for the UniLab II

8031 and 8051 piggyback

ORION Instruments, Incorporated
702 Marshall Street
Redwood City, California
94063

Contents

Features 3

Cable Connections 5

Disassembler 8

Gaining DEBUG Control with RB 10

NMI Features 13

Setting up IRQ for NMI Functions 15

DEBUG Operations 16

Moving the Overlay Area 21

Multiple Break Points 22

On-Line Assembler 23

Demo Program 26

Troubleshooting Hints 33

How DEBUG Works 35

Glossary 36

ORION INSTRUMENTS, INC.
8051 and 8051 piggyback Application Notes
for the UniLab II

These two software versions support the 8051 family. The **DDB-51** is used with the 8051 or 8052 in the expanded mode (external rom) the 8031 and 8032, and the 80C31. The **DDB-51P** supports the OKI piggyback M80C51VS in the internal rom mode (via the piggyback socket). The 8051 transparent DEBUG program is shipped installed in a UniLab system file, along with the 8051 disassembler. It provides all of the additional DEBUG functions normally supplied by microprocessor emulation without consuming more than a few bytes of target memory at the high end of program memory (FFC0 and up in the 8031, and FC0 and up in the piggyback).

The Orion DEBUG software package for the UniLab II provides the capabilities of expensive hardware processor emulators with a number of additional features. It is often the first course of action for the engineer to attack a complicated problem by setting a breakpoint somewhere in memory. Then by examining registers, setting them to new values, and single-stepping through the code, the source of the problem may be isolated.

There are more direct methods available to the user of the UniLab II. If the symptom can be described, then the analyzer can usually track down the cause of the problem almost instantaneously. For instance, if the stack is being nested too deeply, the analyzer can be set to trigger on a small address range just below the maximum depth of the stack. By setting the trigger to occur at the end of the trace buffer (with **NORMB**), the events leading up to the stack problem can be displayed. By setting the trigger to happen at the top of the trace buffer (with **NORMT**) events after the trigger are shown. Setting the trigger to be in the middle of the trace buffer (with **NORMM**) allows the inspection of code leading up to the event ..... as well as the course of the program flow after it!

Illegal data values sent to I/O, instructions occuring outside the range of program memory, or overwriting program areas can be easily set to trigger the analyzer, displaying the sequence of events which led up to these unexpected operations. The flow of the program can be captured and compared with the source code. A static image of the contents of the registers may provide only a partial view of the problem.

For this reason, it is advised that you explore the many resources available with the UniLab II. This write-up covers all the processor-specific features of the disassembler/DEBUG, but this is only a small portion of the potential that is available for tracking down those difficult hardware and software bugs that creep unexpectedly into the most carefully designed projects.

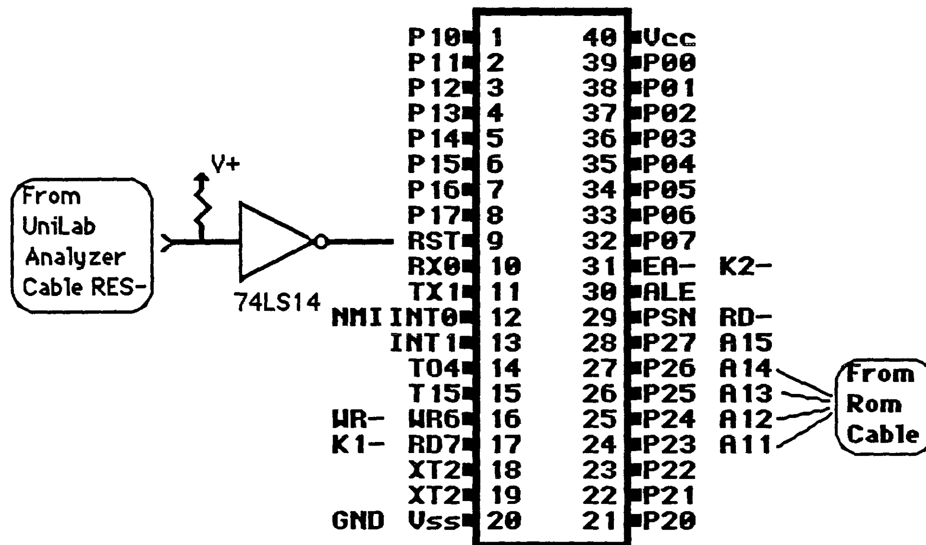
Setting breakpoints is a trial-and-error approach.... It is useful if you can guess at the proper place to inspect the register contents, but it is an indirect approach when compared to the ability to describe the symptom and then to analyze the flow of the program leading up to the problem!

The UniLab's bus state analyzer eliminates the need for most traditional breakpointing and single-stepping in program analysis. However, there are still times when it is useful to see or change register contents at certain points in the program execution. For this reason, the 8051 DEBUG makes use of the internal logic of the UniLab's ROM emulator/analyzer to implement all of the standard DEBUG functions. You can set breakpoints, examine and change both memory and register contents in the target system, and single-step through your program.

The DEBUG package includes a help screen which displays both general UniLab disassembler/DEBUG words as well as target-specific commands. Type **HO** to display this on-line help screen.

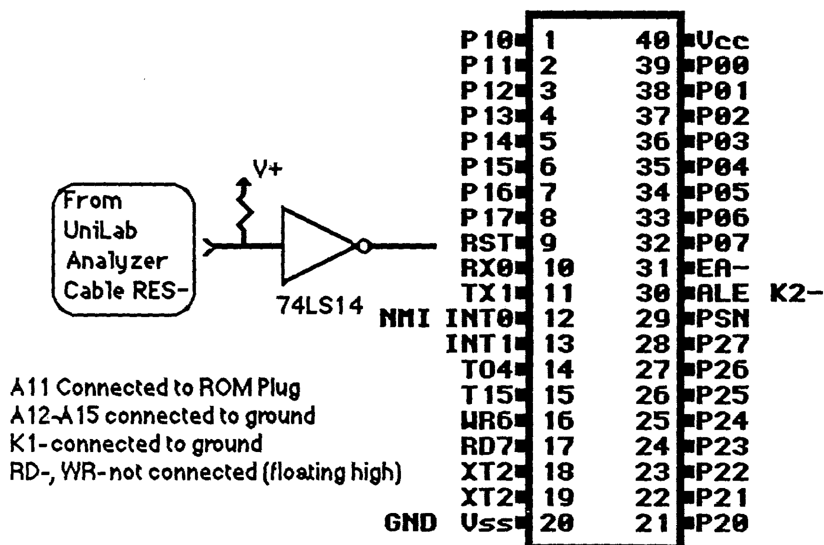
Cable Connections

Here are the cable connections required for the expanded (external rom) hook-up to the UniLab (note that the low order address lines and the data lines are connected via the ROM plug which is plugged into the external ROM socket):

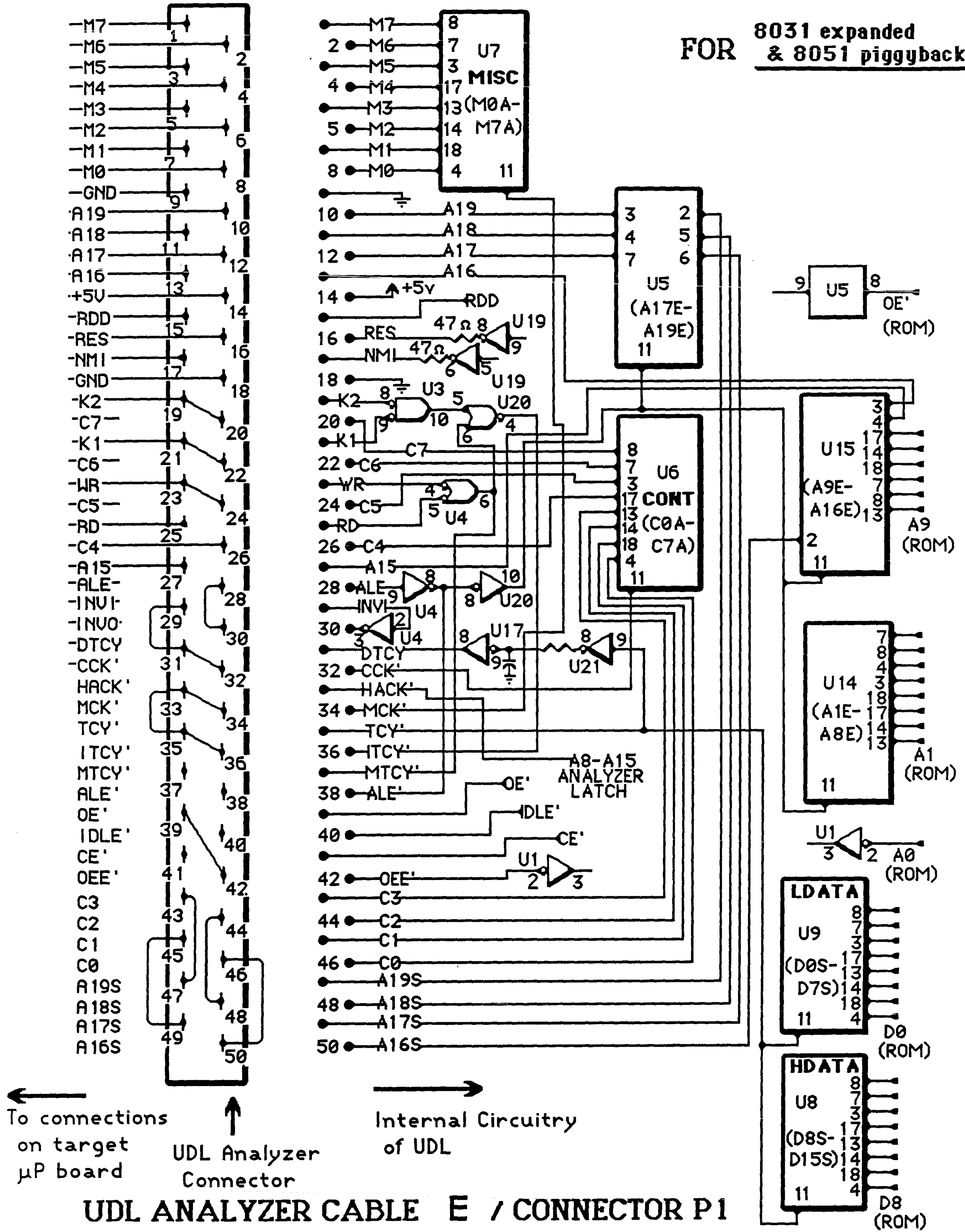


Here are the cable connections required to the piggyback chip from the UniLab. The ROM cable must be plugged into the piggyback connector, A11 must be connected to the rom plug, and A12 through A15 are grounded via the 6-pin jumper included with the UniLab.

E Cable 8051 Piggyback E cable



CABLE E



UDL ANALYZER CABLE E / CONNECTOR P1

Disassembler

At any time you can switch the trace disassembly off and on by entering **DASM'** for a HEX format trace display or **DASM** for disassembled format. You can also disassemble directly from the emulated ROM image by entering **adr n DM** where adr is the target address and n is the number of lines desired.

To disassemble from address 30 9 lines of code you just enter **30 9 DM** . A sample display is shown below:

```
LTARG.JMP  0030  758168   MOV SP,#68
           0033  7412     MOV A,#12
           0035  7834     MOV R0,#34
           0037  7982     MOV R1,#56
           0039  7A83     MOV R2,#78
           003B  7B9A     MOV R3,#9A
           003D  7C04     MOV R4,#4
           003F  04       INC A
           0040  04       INC A
```

Note that a symbol has been defined for 0030 (LTARG.JMP) and the built-in symbol for 81 (SP) is automatically displayed when symbols are turned on.

The **DM** display can be made to pause by pressing any key. It will resume if you press another key, or abort if you press a key twice in rapid succession during a pause. Note that this is the same technique used in the **MDUMP** display.

The cycle types on the 8051 allow the following macros to be defined (external rom mode required):

```
FETCH  70 TO 7F CONT
READ   30 TO 3F CONT  ( not on piggyback chip - Read shows same as Fetch)
WRITE  50 TO 5F CONT
```

The 8051 disassembler automatically skips extra fetches which occur during instruction execution, and pre-fetches which occur before branches. Since these extra cycles are predictable, logic is included in the disassembler for hiding them.

Remember that if the disassembler is not in sync, cycles will be hidden which are not extras. This problem is more apparent in the piggyback version, since instruction fetches cannot be identified by the CONT column. You can look at all cycles by turning the disassembler off with **DASM'** .

To make sure the disassembler is in sync, always start the trace at least a few cycles before the area of interest so there is time for the disassembler to fall into sync. You can start disassembly at the correct point in the trace by entering **n TN** where n is the first cycle of an instruction. You can look at all cycles by simply turning off the disassembler with **DASM'** . External read and write cycles can be indentified in the external rom mode. The piggyback chip, however, has no hardware indication of the cycle type. Use **n TN** to start the trace display at a hidden cycle which may be the actual first cycle of the opcode if the display looks erratic.

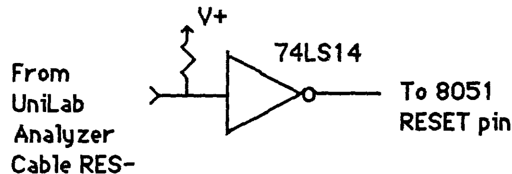
Don't try to trigger the analyzer on the address immediately after a conditional branch, since that address will be pre-fetched even if the branch does occur. (Breakpoints at that address are OK however as they will be effective only if they really execute.)

The internal registers of the 8051 are assigned symbolic names automatically when the symbol table is turned on (by entering **SYMB**, loading a symbol file, or defining a new symbol with **IS**). These are local two-way symbols similar to the symbols in the main UniLab program. The difference is the 8051 local symbols are always present, and are not stored or saved with the UniLab main symbol file. The symbol names can be typed in as arguments for UniLab commands. Here are the assigned symbols:

Address	Name	Address	Name	Address	Name
80	P0	81	SP	82	DPL
83	DPH	88	TCON	89	TMOD
8A	TL0	8B	TL1	8C	TH0
8D	TH1	90	P1	97	PCON
98	SCON	99	SBUF	A0	P2
A8	IE	B0	P3	B8	IP
C8	+T2CN	CA	+RCP2L	CB	+RCP2H
CC	+TL2	CD	+TH2	D0	PSW
E0	ACUM	F0	BREG		

Establishing DEBUG Control with RB

The reset circuit on the 8051 is non-standard, so the UniLab's open-collector RES- output on the analyzer cable will not work if connected directly. We suggest that you build the adapter circuit below, or add the components to your prototype system (DEBUG will not work without a working automatic reset.):



To first activate DEBUG via software you simply set a breakpoint by entering **RESET n RB** where n is the address of your first breakpoint. Since this is a software breakpoint, DEBUG insterts an opcode at that address to be executed. This address must be the first byte of an instruction. DEBUG uses the stack also, so don't set a breakpoint at an instruction before initializing the stack pointer. Though you can set breakpoints in target RAM once DEBUG control is established, the first software breakpoint must be in emulated ROM.

If the breakpoint is reached, you will see a display of all target register contents and a disassembly of the next instruction to be executed (the present breakpoint address). If the breakpoint address is never reached, you can get immedate DEBUG control by hitting the carriage return to execute **NMI** (see next section). If you get an "**NG**" message, see the end of this write-up. While you are stopped at a breakpoint, you can change register contents or target memory . You can then continue to the next breakpoint at address n by entering **n RB**. You can also go to another part of the program at address m before breaking at n by entering **m n GB**.

If you want to exit DEBUG you can Go to location m and let the program run without DEBUG control by entering m **G**. If you enter **m GW** (Go and Wait) the program will resume from location m next time the analyzer is started. After you have done either of these commands, the only way to regain DEBUG control is to use **RESET n RB**.

The display below is an example of a breakpoint display. It was obtained by entering **RESET 40 RB** with the LTARG program:

```
A=13 PSW=01(cafbbv-P) R0=34 R1=56 R2=78 R3=9A R4=04 R5=03 R6=FF R7=FF
 DPTR=   0 SP=68 IE=60
0040  04        INC A              (next step)  ok
```

Note that the flag register bits are displayed as letters just to the right of the hex contents of the flag (PSW) register. Capital letters indicate that a flag is set while a small letter indicates that it is reset. The register contents displayed are the contents before the instruction at the breakpoint address is executed. Entering **N** (Next) will execute the disassembled instruction at the breakpoint location (0040) and give a new register display with the next location (0041) disassembled.

The 8 working registers and A, PSW, SP, DPTR, and IE are displayed automatically at the breakpoint. If you are not changing register banks much, you can assign mnemonic names to R0-R7. For example, if you enter **6 RNAME USER#** the breakpoint display will print USER# instead of R6.

The **N** command does a "next step" of the target program. It actually executes **RB** (Resume to Breakpoint) with the breakpoint automatically set at the start of the next instruction in sequence (just after the one which is disassembled). This is similar to single stepping except when the next instruction is a branch. If you want to follow a branch you must enter **adr RB** where adr is the address the branch will take.

If you use **N** to single step through a loop, you will only see one pass through the loop, then you will see the first step after exiting the loop. This is often more convenient than conventional single stepping and can save you time in many cases, because you don't have to take any action to go once through status and delay loops which might otherwise execute 1000's of times.

Another loop-related DEBUG feature feature is the **LP** loop command. If you are stopped at a breakpoint which is part of a loop, simply enter **LP** to go around the loop once and stop at the same point (you must not be at the last step in the loop). If you want to resume the program when you are sitting at a breakpoint, just enter **RZ** and you'll be back in the program from where you left off.

The **LP** command is actually a macro which executes **N** internally then sets a breakpoint at the previous address. You will see the "(next step)" of the **N** executing to remind you of the internal workings of the **LP** function. The software breakpoint opcode used in the 8051 is actually a three-byte absolute JMP instruction. Therefore, if the next instructions after the breakpoint are single-byte instructions, the processor must advance far enough ahead so the the breakpoing set by **LP** doesn't "step on" the next instruction to be executed. You will see the "(next step)" displayed for each internal **N** that **LP** needs to do.

Be aware of this, since **N** cannot be executed at the end of a loop. If you do this, a software breakpoint will be inserted at the address after the loop. If this is code, you may never get back in the loop again, and **LP** will just wait forever. If the next byte is a data area, you will not get DEBUG control either, since the software breakpoint will never be executed.

The NMI Features

DEBUG control is established with **RESET adr RB** (via a software breakpoint) or with the **NMI** command if you have the NMI connection from the analyzer connected to the 8051 IRQ pin. Gaining DEBUG control this way is dependent upon the UniLab strobing the IRQ pin of the 8051 and causing the program to vector to our own DEBUG routines via the IRQ vector.

The 8051 DEBUG has an interrupt breakpoint feature which allows you to use this NMI output of the analyzer as a substitute for software breakpoints. You must have the NMI connection from the analyzer cable to one of the IRQ pins of the 8031. Make sure that your target board has nothing connected to this pin. It is usually connected to +5 volts if unused. If you have a driver, you can connect the UniLab NMI to an input of the driver. Make sure that no other interrupt's will occur on this pin to cause triggering on other conditions.

You can establish breakpoint control on a working program (with functioning stack) by just pressing the **F4** function key at any time. Entering **NMI** or hitting the **F4** again, will cause the 8051 to advance to the next instruction. This is a true single-step mode which will follow branches and loops. The **N** command can be very useful since you can follow program flow without having to single step through every iteration of a loop. The **NMI** single-step is useful if you want to follow branch or jump instructions. Single stepping is mainly useful when a complex algorithm is being executed and you want to keep track of register contents at each program step. Usually the analyzer gives you a much better picture of overall program flow by showing you a "snapshot" of the actual execution of the program in real time.

All of the other DEBUG commands such as **N**, **RB**, **G** can be used along with the **NMI** key. If you use **RB** to set a breakpoint that never arrives, you can press any key to give up waiting and the program will automatically regain breakpoint control using the NMI signal. " **-nmi-** " will be displayed whenever this happens, or when you press the **F4** key.

You can also use the power of the analyzer truth-table logic to breakpoint on complex functions of data, address ranges, etc. To use the feature, enter **RI** followed by a trigger spec, then **SI** will release the program till trigger occurs. For example,
RI F800 TO F900 ADR SI
will stop the program and display the registers after an address in the range of F800 to F900 is accessed. Note that you cannot use qualifiers or pass counts with this feature since it uses the qualifier truth table as part of the internal workings of DEBUG.

The patches necessary for DEBUG operation are automatically patched in whenever automatic reset occurs. You can disable this feature if it causes problems by entering **NMIVEC'** and re-enable it by entering **NMIVEC.**

You can also use the full power of truth tables in the triggering logic to interrupt the target processor when the analyzer enters the trigger state. You can set up your own interrupt routine to be executed when this occurs. For example, **NORMT INT AFTER NOT F800 TO F9FF ADR S** will interrupt the target processor (via the NMI-output) one cycle after the program goes outside of the F800 to F9FF range. This can be very useful for working on systems where program bugs can "self-destruct" peripherals. All you have to do is write an interrupt routine which gently shuts things down. In order to use this feature, you must disable the NMI routines of the 8051 DEBUG by entering **NMIVEC'**. This will allow you to set the address of your own diagnostic routine at the IRQ vector.

Setting up IRQ for NMI functions

The program loaded by **LTARG** has four lines of code for setting up the IRQ on the 8031. Your target program must have these lines as well in order for the **NMI**, **RI** and **SI** commands to function.

Here is a fragment of the 8031 LTARG program, with the required code underlined:

```
Adr   Opcode   Mnemonic

0039  7A78     MOV R2,#78
003B  7B9A     MOV R3,#9A
003D  7CBC     MOV R4,#BC
003F  7DDE     MOV R5,#DE
0041  D2B8     SETB B8        ( set up IP for external interrupt0)
0043  D2A8     SETB A8        ( enable IE for INT0)
0045  D2AF     SETB AF        ( enable IRQ's )
0047  D288     SETB 88        ( set INT0 to be edge-triggered)
0049  04       INC A
```

There are two possible IRQ's to use on the 8031, pin 12 which is called INT0 or pin 13, INT1. The two vector addresses are at 0003 and 0013.

You can specify **IT0** or **IT1** to choose one or the other. INT0 is the default. If you choose INT1 you will have to connect the NMI line to pin 13, and put the following four lines into your program, in place of the four listed above:

```
Opcode   Mnemonic

D2B8     SETB B8
D2AA     SETB AA       ( enable IE for INT1)
D2AF     SETB AF
D28A     SETB 8A       ( set INT1 to be edge-triggered)
```

DEBUG Operations

Using **RB**, **NMI**, or **RI** <trigger spec> **SI** will get DEBUG control of the 8051/31 While you are stopped at a breakpoint, you can change register contents. For example, **12 =R3** will change the R3 register contents to 12. (Note that there is a space after the 12 but no space after the equal sign.) Any of the registers shown in the breakpoint display can be changed with similar commands. Type **HO** to see a list of available 8051/31 DEBUG commands.

You can also read and write target RAM with all of the DEBUG commands normally used on emulation memory. You can use **M, MM, ORG, M!, MM!, M?, MM?, MD, MDUMP, MMOVE, MCOMP,MFILL, MLOAD, MSAVE, MLOADN, BINLOAD, BINSAVE, HEXLOAD**, and **HEXRCV**, just as you would on emulator memory. See the UniLab manual for a full explanation of these generic DEBUG functions.

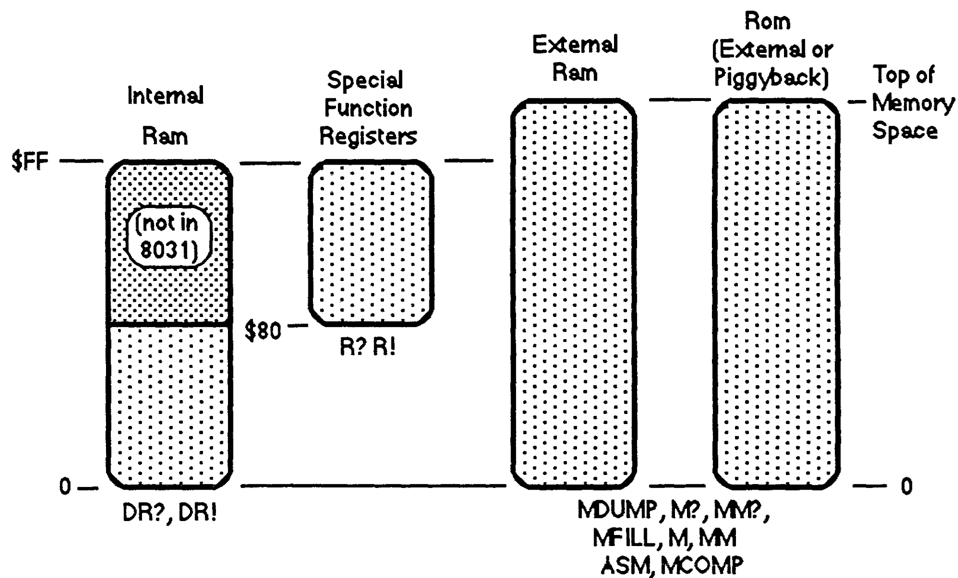
Whenever these commands are used, the UniLab program checks the address to see if it is enabled (set by the **EMENABLE** command) as an emulated ROM address. If it is not enabled, it will attempt to execute the command on the target RAM. If DEBUG is not loaded, or DEBUG is not in control, an error message will so indicate. If DEBUG control has been established, you'll get a message:

```
target memory (not EMENABLEd)
```

to let you know that this is target memory rather than emulation memory, and that DEBUG is doing the operation. This is not an error message. It is just a reminder that the DEBUG function is in charge of the operation, and that this is not an exchange of data between the host and emulation memory (which can occur even without the target processor connected). Information can be moved back and forth between target RAM and emulated ROM with the **MMOVE** command, but remember that the decision as to which memory type is being acted upon is made only once, at the start of the operation.

The Special Function Registers can be altered by entering **<data> adr R!** or displayed by entering **adr R?** . All of the named locations such as P0, IP, TH0, etc are defined as symbols, so you can, for example, look at IP by entering **IP R?**. If you enter **23 P0 R!** you will write 23 to port 0. You should always use the specific DEBUG commands **=DPTR**, **=IE**, etc. to change any of the special function registers that have specific commands. This is because DEBUG uses some of these internally and if you look at them directly with **R?** or try to change them with **R!**, you may not get proper values.

Here is a memory map of the four memory areas available in the 8051 with the associated UniLab commands for examining and changing memory.



Again, dont' use **DR?** or **DR!** to inspect or modify registers that have special commands, such as **=R0**, and **=R1**. These are used internally by DEBUG and may not show the same values as the breakpoint display. Also note that **R?** and **R!** will work just like **DR?** and **DR!** in the area from 0 to $7F. **DR?** and **DR!** are mainly useful in the 8032.

If you have external memory, you can use all of the commands used on emulation memory such as **MDUMP**, **MFILL**, **MMOVE**, **M?**, **MM?**, **M!**, and **MM!** and they will go to external memory if the address isn't enabled for emulation. Typing **TRAM** will allow you to use **MDUMP**, **M?**, **MM?**, **MFILL**, **DM** , **M!**, **MM!**, and **MCOMP** on external ram at the same address as emulated memory.

Typing **TRAM** allows you to use **MDUMP**, **M?**, and **MM?** on any external ram which has the same address as emulation memory. Typing **TRAM'** resets to normal use of **MDUMP**. After entering **TRAM'**, you can still use **R?**, **R!**, **DR?**, and **DR!** to examine and change internal registers and memory. **TRAM'B** has no effect on these commands.

When DEBUG is active , a short routine is patched into the target program starting at location FFC0 (FC0 on the piggyback) when the target is started up with RESET enabled . These are the only reserved locations except for the IRQ vector at 0003 or 0012. Overlays are installed above location FFC0 and automatically restored. You can thus use the area above FFC0 but most DEBUG operations won't work properly in the locations above this address. These bytes at the top of memory are required by DEBUG functions for saving and restoring interrupt enable status, and returning to the program. This memory space must be enabled by entering **ALSO FF00 EMENABLE** after the EMENABLE statements you would normally make for your system. (The memory locations used by the 8051 piggyback disassembler/DEBUG package are at FC0 to FC7.)

The DEBUG sets a breakpoint by patching a 3-byte LCALL instruction into your program, so be aware that if the program tries to jump to or use either of the 2 bytes following the breakpoint there will be trouble. Each time a new breakpoint is set the previous breakpoint addresses are restored to their original contents.

The overlay area is restored at the end of each DEBUG operation but the DEBUG commands may not work properly on data stored in the overlay area. For example, you can generally move data blocks from the emulated ROM to the target RAM, but not if the block includes the overlay area (FFC0-FFF8 on the 8031, FC0-FFF on the piggyback 80C51).

The vectors in this rerved area are automatically patched when the analyzer starts up and RESET is enabled. If you want this vector to be disabled, type **RSP'**. This will make DEBUG inoperative, but allow you to use the analyzer in a completely transparent mode, not affecting the target memory in any way. To re-enable DEBUG, type **RSP**. To disable the **NMI** features only (and the automatic patching of the IRQ vector), type **NMIVEC'**. To re-enable, type **NMIVEC**.

Remember that your program is essentially stopped when you are at a breakpoint, so if you have a watchdog timer in your system it must be disabled to prevent extra resets. Interrupts are disabled while stopped at a breakpoint but re-enabled every time a program step is executed.

The 8051 DEBUG has an "auto-breakpoint" capability. This nearly transparent function allows you to inspect or modify registers or ram without first gaining DEBUG control. DEBUG will assert an automatic NMI (seen by a " -nmi -"), then display ram or register or write to ram or register, and then let the target program resume. All this is done without having debug control asserted explicitly using **RB** or **NMI**. This feature allows the use of all memory modification commands such as **MFILL, M!**, etc. It will not work with register modifying commands such as **=R0**, or **=DPTR**.

In addition to the normal breakpoint display, you can display anything you want by using the **BPEX** command. For example, if you want to automatically dump external RAM at locations 00-3F at each breakpoint, you first define a macro that will do this by entering the following:

: PDUMP TRAM' 0 3F MDUMP TRAM ;

If you now enter **BPEX PDUMP** the macro (which we called **PDUMP**) will automatically execute after the register display whenever a breakpoint is reached. The macro definition is started by entering the colon (**:**) , then its name is defined. The words following the macro name (in this case **PDUMP**) are UniLab words just as you would enter from the command mode. The macro is terminated by the semicolon (;) . When the name of the macro is typed, all of the words in its definition will be executed just as if they had been typed. A second macro can be added by defining another word and linking it to DEBUG with **BPEX2**. When the macro is linked to DEBUG with the **BPEX** or **BPEX2** command it becomes an extension to DEBUG. If the system is saved to disk with **SAVE-SYS**, this extension will be permanent. Note that we restored the **TRAM** mode after the **INTRAM** change to allow DEBUG to work in the normal rom space for the next DEBUG command. If you want to eliminate this automatic memory window display, just enter **BPEX NOOP** to install the **NOOP** command (which does nothing) in place of **PDUMP**.

Moving the Overlay Area

The command **=OVERLAY** can be used to change the location of the overlay area. It expects one address, such as

1234 **=OVERLAY**.

The system must be saved with **SAVE-SYS** <name> for the change to become permanent.

The overlay can only be placed at an even address. The word **=OVERLAY** takes care setting this to the proper boundary

When changing the overlay area, you must **EMENABLE** that range of memory, and make certain that there is enough room above this address in emulated memory for DEBUG code to be overlaid. The best guideline is to leave the lower 8 bits of the existing overlay area the same and only change the upper 8 bits. For instance, if the overlay area is default at FFC0, the user shouldn't change it to CFF0 if D000 and up is not emulation memory. The wisest choice would be to set it anywhere from CF00 to CFC0. Keep the overlay in a single "page" of emulation memory.

LTARG is set up so that if the user changes the overlay area, **LTARG** will also automatically **EMENABLE** that 2K block of memory in addition to its regular run space.

The **HO** screen shows the current location of the overlay area, reflecting either the default area or the new one changed by **=OVERLAY**.

=OVERLAY will initialize the UniLab, then show the new settings. If the UniLab is not hooked up, the re-configuration will still take effect. If you hit the **CTRL-BRK** keys when the unit tries to initialize, then save the system, it will be re-configured the next time the system is started up.

MULTIPLE BREAKPOINTS

Up to 8 additional breakpoints can be set by using the **SMBP** command. The format of this command is **adr bp# SMBP** so entering **234 2 SMBP** will set breakpoint number 2 to address 234:

```
234 2 SMBP ( command entered by user to set breakpoint #2 at address 1234)
1 ----   2 0234   3 ----   4 ----   5 ----   6 ----   7 ----   8 ----    ok
```

Whenever **SMBP** is used, all 8 breakpoints are displayed. You can display the breakpoints with **DMBP** . You can clear any single breakpoint by entering **bp# RMBP** , or clear all 8 with **CLRMBP** .

```
678 4 SMBP ( set another breakpoint at $5678, assigning it as multiple breakpoint #4)
1 ----   2 0234   3 ----   4 0678   5 ----   6 ----   7 ----   8 ----    ok

468 5 SMBP ( ...and assign #5 at 2468)
1 ----   2 0234   3 ----   4 0678   5 0468   6 ----   7 ----   8 ----    ok

2 RMBP   ok ( Remove breakpoint #2)

DMBP ( display all current multiple breakpoints)
1 ----   2 ----   3 ----   4 0678   5 0468   6 ----   7 ----   8 ----    ok
```

Use the **SMBP** command to set all but one of the breakpoints, then use **RB** or **GB** to set the final breakpoint. The breakpoints set with **SMBP** are "sticky", unlike the breakpoint set by **RB** or **GB** . They remain set even after you are stopped at a breakpoint. You can use **RMBP** to clear the breakpoint you are stopped at and then use **RB** or **GB** again.

You can use **N** after the breakpoint to single step, but it will automatically clear all breakpoints. Using commands which start the analyzer such as **S**, **STARTUP**, or **NOW?** will also clear all breakpoints. You can use **DMBP** to check the settings.

Before you decide to use multiple breakpoints, give some thought to the possibility of using the analyzer to do the same job. Generally the analyzer is a faster, more efficient tool for finding errors in program flow.

ON-LINE ASSEMBLER

(UniLab systems only)

Using **ASM** or **ASM-FILE** you can assemble code directly into emulation ROM. You can also assemble into target RAM after you have established breakpoint control. The number base is always hexadecimal.

The on-line assembler uses the standard instruction set for this processor except that we do not support the location counter symbol, "*". There must be at least one space between opcode and operand. When using numbers, no suffixes or prefixes are necessary since you are always in hexadecimal.

The assembler scans an instruction in the following order to solve any ambiguities.

1) register name
2) symbol
3) user defined registers
4) FORTH constant
5) number

If your instruction set uses register A and you wish to use A for a numeric value, you should type in 0A to avoid ambiguity.

Instructions can be up to 40 characters long excluding spaces. Arithmetic expression can be 30 characters long. Symbols are limited to 16 bit numbers.

When **ASM** is invoked, it sets the variable for origin. Origin is the address used by **M** and **MM** command. You can type **100 ASM** and even if there is a parse error, the origin will still be set to the new value.

8051 Instruction Set

```
* ACAL   pp             INC    DPTR              MOVEX  A,@R1
  ACALL  pp                    reg               MUL    AB
  ADD    A,#nn                 nn                NOP
         A,@R0          JBC    nn,rr             ORL    A,#nn
         A,@R1          JB     nn,rr                    A,@R0
         A,reg          JC     rr                       A,@R1
         A,nn           JMP    @A+DPTR                  A,reg
  ADDC   A,#nn          JNB    nn,rr                    A,nn
         A,@R0          JNC    rr                       C,/nn
         A,@R1          JNZ    rr                       C,nn
         A,reg          JZ     rr                       nn,#nn
         A,nn         + LCAL   nnnn                     nn,A
  AJMP   pp             LCALL  nnnn              POP    nn
  ANL    A,#nn          LJMP   nnnn              PUSH   nn
         A,@R0          MOV    @R0,#nn           RET
         A,@R1                 @R1,#nn           RETI
         A,reg                 @R0,A             RL     A
         A,nn                  @R1,A             RLC    A
         C,/nn                 @R0,nn            RR     A
         C,nn                  @R1,nn            RRC    A
         nn,#nn                A,#nn             SETB   C
         nn,A                  A,@R0                    nn
  CJNE   @R0,#nn,rr            A,@R1             SJMP   rr
         @R1,#nn,rr            A,reg             SUBB   A,#nn
         A,#nn,rr              A,nn                     A,@R0
         A,nn,rr               C,nn                     A,@R1
         reg,#nn,rr            DPTR,#nnnn               A,reg
  CLRA                         reg,#nn                  A,nn
  CLRC                         reg,A             SWAP   A
  CLR    nn                    reg,nn            XCH    A,@R0
  CPL    A                     nn,#nn                   A,@R1
         C                     nn,@R0                   A,reg
         nn                    nn,@R1                   A,nn
  DA     A                     nn,A              XCHD   A,@R0
  DEC    @R0                   nn,C                     A,@R1
         @R1                   nn,reg            XRL    A,#nn
         A                     nn,nn                    A,@R0
         reg            MOVC   A,@A+DPTR                A,@R1
         nn                    A,@A+PC                  A,reg
  DIV    AB             MOVX   @DPTR,A                  A,nn
  DJNZ   reg,rr                @R0,A                    nn,#nn
         nn,rr                 @R1,A                    nn,A
  INC    @R0                   A,@DPTR
         @R1                   A,@R0
         A
```

* same as ACALL

\+ same as LCALL

#nn	8-bit immediate
nn	8-bit number
nnnn	16-bit number
rr	8-bit relative address
pp	11-bit page address
reg	R0 through R7

On-Line Assembler Error Messages

FILE ACCESS ERROR -- Assembler cannot find the table file. The table file has a "8051.TBL" file extension. The assembler expects the table file to be in the directory where ORION is set to. Have you renamed the table file?

PARSE ERROR -- Check instruction set to make sure you have right syntax. If you are using symbols, make sure they are defined. (Try typing in the symbol name. If "not recognized", the symbol is not defined.) Maybe the instruction or arithmetic expression is too long.

NOT ENOUGH MEMORY -- Assembler cannot allocate a large enough segment to read in the table file. Free up some memory using **=HISTORY** or **=SYMBOLS**.

BAD TABLE FILE -- The table file is corrupted. Copy "8051.TBL" file from your master diskette.

CAN'T R/W NON-EMULATED ADDRESS WITHOUT WORKING DEBUG CONTROL!
-- Assembler uses the **M** command to write to emulation memory or target RAM. Either memory has to be enabled or DEBUG control must be established.

TARGET MEMORY NOT EMENABLED -- This is NOT an error message - just an indication that DEBUG is doing the operation in target RAM rather than emulation memory.

8051 DEMO PROGRAM

Below you will find a trace of a built-in program demonstrating the 8051 DEBUG operation. The **LTARG** command enables memory and also loads the sample program. The program simply loads known values into the registers, sets up the stack, then increments a register for a while before jumping back to the start of the program. Since this command also sets up all enables and DEBUG parameters correctly, you should try using DEBUG with it if you have a problem with your own software. If DEBUG works with the **LTARG** program but not with yours, try to examine the differences between the two programs in light of the specific requirements for your target system.

Text entered from the keyboard by the user is underlined, and comments are small italicized remarks in parenthesis. All other display is from the UniLab program, and should be similar to your display. After executing a command, the program usually responds with an **ok** message.

```
>UL51       ( logon to UniLab system...)

                         UniLab                   Copyright 1986
                           II                  Orion Instruments
                       Version 3.20              Redwood City, CA

 8051 disassembler installed - with DEBUG.
 Emulator Memory Enable Status:
               F =EMSEG
       0 TO  7FF EMENABLE
ALSO F800 TO FFFF EMENABLE ( note area enabled for overlay)


Initializing UniLab...
  Initialized


LTARG       ( Load in sample program, set up emulation memory)
 Emulator Memory Enable Status:
               F =EMSEG
       0 TO  7FF EMENABLE   ( enable rom area)
ALSO F800 TO FFFF EMENABLE  ( enable overlay area)

STARTUP resetting ( reset target, show first cycles of operation)
 cy# CONT ADR DATA                             HDATA    MISC
   0  7F 0000 020030   LJMP 30            11111111 11111111
   4  7F 0030 758168   MOV 81,#68         11111111 11111111 ( set stack ptr)
   8  7F 0033 7412     MOV A,#12          11111111 11111111
   A  7F 0035 7834     MOV R0,#34         11111111 11111111
   C  7F 0037 7956     MOV R1,#56         11111111 11111111
   E  7F 0039 7A78     MOV R2,#78         11111111 11111111
  10  7F 003B 7B9A     MOV R3,#9A         11111111 11111111
  12  7F 003D 7C04     MOV R4,#4          11111111 11111111
  14  7F 003F 04       INC A              11111111 11111111
  16  7F 0040 04       INC A              11111111 11111111
  18  7F 0041 04       INC A              11111111 11111111
  1A  7F 0042 04       INC A              11111111 11111111
  1C  7F 0043 04       INC A              11111111 11111111
  1E  7F 0044 04       INC A              11111111 11111111
  20  7F 0045 04       INC A              11111111 11111111
  22  7F 0046 04       INC A              11111111 11111111
  24  7F 0047 04       INC A              11111111 11111111
  26  7F 0048 04       INC A              11111111 11111111
  28  7F 0049 04       INC A              11111111 11111111
  2A  7F 004A 04       INC A              11111111 11111111
  2C  7F 004B 04       INC A              11111111 11111111
  Pg Dn (trace resume)  Home (top) n TN (from step n)  T (from n=-5 ) ok

RESET 40 RB resetting ( set a breakpoint at address $0040, reset target
                        and run until we break to get DEBUG control)
A=13 PSW=01(cafbbv-P) R0=34 R1=56 R2=78 R3=9A R4=04 R5=03 R6=FF R7=FF
 DPTR=    0 SP=68 IE=60
0040  04       INC A            (next step)  ok
```

N *(execute next program step, note A register is incremented)*

```
A=14 PSW=01(cafbbv-P) R0=34 R1=56 R2=78 R3=9A R4=04 R5=03 R6=FF R7=FF
 DPTR=    0 SP=68 IE=60
0041  04        INC A              (next step)  ok
```

27 =R6 ok *(Set register 6 to a new value of $27)*

R *(Display registers, note R6)*

```
A=14 PSW=01(cafbbv-P) R0=34 R1=56 R2=78 R3=9A R4=04 R5=03 R6=27 R7=FF
 DPTR=    0 SP=68 IE=60
0041  04        INC A              (next step)  ok
```

TRAM ok *(switch DEBUG operations for external ram)*

40 4F MDUMP target adr (not EMENABLEd) *(look at external ram..)*

```
    40    A2 A2 A2 A2 A2 A2 A2 A2  A2 A2 A2 A2 A2 A2 A2 A2    ................
 ok
```

40 ORG ok *(start patching in external ram at $0040)*

1 M 2 M 3 M 4 M 5 M 6 M 7 M 8 M target adr (not EMENABLEd)
(store values of 1,2,3...8 at $0040 to $0047)

40 4F MDUMP target adr (not EMENABLEd) *(display results in external ram)*

```
    40    01 02 03 04 05 06 07 08  A2 A2 A2 A2 A2 A2 A2 A2
................
```

27 48 M! 347 49 MM! target adr (not EMENABLEd) *(store a $27 at address $0048 and $0347 at address $0049,A)*

40 4F MDUMP target adr (not EMENABLEd) *(inspect results)*

```
    40    01 02 03 04 05 06 07 08  27 03 47 A2 A2 A2 A2 A2    ........'.G.....
```

43 M? target adr (not EMENABLEd) **4** ok *(look at just one byte at $0043)*

44 MM? target adr (not EMENABLEd) **506** ok *(inspect word at $0044,45)*

TRAM' ok *(select emulation memory as normal memory area)*

34 41 DR! 35 42 DR! ok *(store $34 into register $41, store $35 intro register $42)*

41 DR? 34 ok *(look at just one register)*

40 4F MDUMP *(dump it out, note **inc a** opcodes from emulation memory)*

```
    40    04 04 04 04 04 04 04 04  04 04 04 04 04 04 04 04    ................
```

```
30 10 DM        ( disassemble $10 lines starting at address $30)
0030  758168    MOV 81,#68
0033  7412      MOV A,#12
0035  7834      MOV R0,#34
0037  7956      MOV R1,#56
0039  7A78      MOV R2,#78
003B  7B9A      MOV R3,#9A
003D  7C04      MOV R4,#4
003F  04        INC A
0040  04        INC A
0041  04        INC A
0042  04        INC A
0043  04        INC A
0044  04        INC A
0045  04        INC A
0046  04        INC A
0047  04        INC A            ok

30 IS LTARG.JMP  ok ( declare a symbolic name for address $30)

30 10 DM  (disassemble again, now symbols are turned on,
            and special function registers will get automatic symbolic labels)
LTARG.JMP  0030  758168    MOV SP,#68      ( note SP label)
           0033  7412      MOV A,#12
           0035  7834      MOV R0,#34
           0037  7982      MOV R1,#56
           0039  7A83      MOV R2,#78
           003B  7B9A      MOV R3,#9A
           003D  7C04      MOV R4,#4
           003F  04        INC A
           0040  04        INC A
           0041  04        INC A
           0042  04        INC A
           0043  04        INC A
           0040  04        INC A
```

NORMT RESET SP DATA S resetting *(use symbol label to set trigger spec)*

```
 cy# CONT                ADR DATA                          HDATA    MISC
  -5  7F                 0000 020030    LJMP LTARG.JMP  11111111 11111111
  -1  7F LTARG.JMP       0030 758168    MOV SP,#68      11111111 11111111
   3  7F                 0033 7412      MOV A,#12       11111111 11111111(note cy# 0
   5  7F                 0035 7834      MOV R0,#34      11111111 11111111 is in middle
   7  7F                 0037 7982      MOV R1,#82      11111111 11111111 of MV sp
   9  7F                 0039 7A83      MOV R2,#83      11111111 11111111 instruction)
   B  7F                 003B 7B9A      MOV R3,#9A      11111111 11111111
   D  7F                 003D 7C04      MOV R4,#4       11111111 11111111
   F  7F                 003F 04        INC A           11111111 11111111
  11  7F                 0040 04        INC A           11111111 11111111
  13  7F                 0041 04        INC A           11111111 11111111
  15  7F                 0042 04        INC A           11111111 11111111
  17  7F                 0043 04        INC A           11111111 11111111
  19  7F                 0044 04        INC A           11111111 11111111
  1B  7F                 0045 04        INC A           11111111 11111111
  1D  7F                 0046 04        INC A           11111111 11111111
  1F  7F                 0047 04        INC A           11111111 11111111
  21  7F                 0048 04        INC A           11111111 11111111
  23  7F                 0049 04        INC A           11111111 11111111
  25  7F                 004A 04        INC A           11111111 11111111
  27  7F                 004B 04        INC A           11111111 11111111
  Pg Dn (trace resume)  Home (top) n TN (from step n)  T (from n=-5 )
```

R *(we're still stopped at a breakpoint, let's see where...)*

```
A=14 PSW=01(cafbbv-P) R0=34 R1=56 R2=78 R3=9A R4=04 R5=03 R6=FF R7=FF
 DPTR=   0 SP=68 IE=60
0041  04        INC A            (next step)  ok
```

PSW R? **1** ok *(examine individual special function registers...)*

SP R? **68** ok

IE R? **60** ok

PCON R? **FF** ok

```
>UL51P      ( here's a trace of the 80C51 piggyback version)

                         UniLab                 Copyright 1986
                           II                Orion Instruments
                       Version 3.20            Redwood City, CA

 80C51 disassembler installed - with DEBUG.
 Emulator Memory Enable Status:
               F =EMSEG
        0 TO  FFF EMENABLE


LTARG                     (note differences from 8051 non-piggyback version)
 Emulator Memory Enable Status:
               F =EMSEG
        0 TO  FFF EMENABLE  ( entire 4K block enabled for piggyback rom)

STARTUP resetting         ( only difference in disassembler listing is in CONT
column)
(from top of buffer)
 cy# CONT ADR DATA                    HDATA   MISC
   0  3F 0000 020030  LJMP 30        11111111 11111111( note control column)
   4  3F 0030 758168  MOV 81,#68     11111111 11111111
   8  3F 0033 7412    MOV A,#12      11111111 11111111
   A  3F 0035 7834    MOV R0,#34     11111111 11111111
   C  3F 0037 7956    MOV R1,#56     11111111 11111111
   E  3F 0039 7A78    MOV R2,#78     11111111 11111111
  10  3F 003B 7B9A    MOV R3,#9A     11111111 11111111
  12  3F 003D 7C04    MOV R4,#4      11111111 11111111
  14  3F 003F 04      INC A          11111111 11111111
  16  3F 0040 04      INC A          11111111 11111111
  18  3F 0041 04      INC A          11111111 11111111
  1A  3F 0042 04      INC A          11111111 11111111
  1C  3F 0043 04      INC A          11111111 11111111
  1E  3F 0044 04      INC A          11111111 11111111
  20  3F 0045 04      INC A          11111111 11111111
  22  3F 0046 04      INC A          11111111 11111111
  24  3F 0047 04      INC A          11111111 11111111
  26  3F 0048 04      INC A          11111111 11111111
  28  3F 0049 04      INC A          11111111 11111111
  2A  3F 004A 04      INC A          11111111 11111111
  2C  3F 004B 04      INC A          11111111 11111111
  PgDn (trace resume)  Home (top) n TN (from step n)  T (from n=-5 )

RESET 40 RB resetting      ( operation is the same as 8031)

A=13 PSW=01(cafbbv-P) R0=34 R1=56 R2=78 R3=9A R4=04 R5=38 R6=0A R7=DD
 DPTR= F00 SP=68 IE=60
0040  04       INC A             (next step)  ok

N
A=14 PSW=01(cafbbv-P) R0=34 R1=56 R2=78 R3=9A R4=04 R5=38 R6=0A R7=DD
 DPTR= F00 SP=68 IE=60
0041  04       INC A             (next step)  ok

27 =R5  ok
```

```
B
A=14 PSW=01(cafbbv-P) R0=34 R1=56 R2=78 R3=9A R4=04 R5=27 R6=0A R7=DD
 DPTR= F00 SP=68 IE=60
0041  04        INC A             (next step)  ok

N
A=15 PSW=01(cafbbv-P) R0=34 R1=56 R2=78 R3=9A R4=04 R5=27 R6=0A R7=DD
 DPTR= F00 SP=68 IE=60
0042  04        INC A             (next step)  ok
```

TROUBLESHOOTING HINTS

IF YOU HAVE EMULATION PROBLEMS

If your target system doesn't work normally under emulation, it probably means that you haven't properly enabled the emulation. If you enter **STARTUP** and look at the very first location to be read (location 0 on the 8051 µprocessor) you can see if the correct data is being presented to the processor. For example, in the trace below the program blows immediately because the data read at location 0 is FF.

```
-1   7F 0000 FF .......interrupted    11111111 11111111
 0   7F 0001 FF .......interrupted    11111111 11111111
```

We can find out what is actually stored at location 0000,1 by entering **0 MM?**. If it is something other than FF the emulator must not be enabled for location 0000. Remember that the enable is a 20-bit address. The F in column 2 of the trace shows the hex value of A19-A16 to be F. This is the normal case for 8-bit µprocessors where A19 through A16 are not connected, so they float high. It is therefore necessary that you enter **F =EMSEG** before making any **EMENABLE** statements. (Note that there must be a space after the F but not after the = .) This command sets the value for the high order address bits A19 to A16 which will allow the emulation memory to be enabled. You can then enable location 0000 by entering **0 EMENABLE** or **0 TO 7FF EMENABLE** .

IF YOU HAVE ANALYZER PROBLEMS

If you get a "NO ANALYZER CLOCK" message when you try **STARTUP** it may mean that your processor is executing a HALT instruction. If you enter **TD** (dump UniLab trace buffer) you can sometimes see the first cycles that did execute to determine why the processor halted before the trace buffer was full. Use an oscilloscope to check that the processor is indeed producing a clock. The UniLab clock logic is shown on the analyzer cable diagram and in the UniLab manual. Most target systems will confine the address decoding to the Chip Enable signal and put a memory read strobe on Output Enable. On systems where this is not done you may have to disconnect from the Output Enable line on the Emulator Cable (which usually picks up the system Output Enable at the ROM socket) to a valid system Output Enable somewhere else on the target board.

IF YOU HAVE DEBUG PROBLEMS

The 8051 DEBUG package includes a resident test program which can be loaded by entering **LTARG**. Since this command also sets up all enables and DEBUG parameters correctly, you should try using DEBUG with it if you have a problem with your own software. If DEBUG works with the **LTARG** program but not with yours, try to examine the differences between the two programs in light of the specific requirements for your target hardware. Look closely at the startup trace, especially the addresses and the contents of memory at those addresses. The trace display is a true "snapshot" of the bus activity and if an address line is connected wrong, or data is wrong, the display can be a very effective diagnostic aid in getting your target system up and running.

If nothing happens when you try to set a breakpoint it may mean that that address is never reached in the program. Press any key to stop the trigger search, then try a breakpoint closer to the (reset) beginning of the program. Remember that the first breakpoint must be in emulated memory. If the program is really reaching the breakpoint you should be able to trigger by entering **RESET n AS** where n is your breakpoint address.

If your first breakpoint results in a **"NG! ..."** message, it means that the proper trace was not obtained after trigger. Usually this means one of the following:

1. The target system stack is not initialized or not working properly. (Look at the trace of a push & pop execution to be sure that the data read back is the same as what was written.--data not valid for piggyback, though.)
2. One or more of the target address lines are not connected or working.
3. One or more of the CONT inputs (C7, C6, etc.) are not connected properly. The one that indicates data direction (read verses write) is particularly important.
4. The overlay area used by DEBUG is not **EMENABLE**d.
5. Target memory is getting on the bus in conflict with the emulator in the overlay area.
6. RESET from the UniLab is not resetting the target. Check the RESET pin of the target µp to see that it really is going low when you type **STARTUP** (or hit **F9**)

HOW DEBUG WORKS

DEBUG makes use of the UniLab's analyzer and idle register hardware to allow all of the normal functions of a processor emulator to be done with universal hardware. When you enter **RESET adr RB** an LCALL is placed at adr and the idle register is enabled with trigger set for the breakpoint address. A small routine for saving the interrupt status and disabling interrupts is also patched in at the breakpoint address.

The processor then resets and the target program runs until it reaches the breakpoint. Just after the patched-in breakpoint routine disables interrupts the idle register begins holding the processor in a loop by feeding it jump instructions which jump to themselves.

Next the memory contents of the overlay area is saved and a short program which puts the contents of each register on the bus is loaded into the overlay area. The analyzer is started again. It releases the idle loop and allows the register saving program to execute. This program ends in a jump to the breakpoint address , again triggering the idle loop. A host program then displays the register contents in a nice formatted display based on the analyzers trace of the register saving program.

When you are stopped at a breakpoint you can look at a trace of what happened before the breakpoint by entering **TD**. Note that the end of the trace shows the breakpoint routine, idling, then the register save routine. The top of the trace buffer may have old trace information in it but the region before the breakpoint is correct. (Note that some versions filter the trace severely enough that the register saving routine will not disassemble correctly.)

You can use the analyzer to observe a breakpoint in action by doing the following:

1. Load your program or enter **LTARG** to load the demo target program.
2. Enter

   RSP' BPINT RESET n SBP n AS

where n is the desired breakpoint address. **RSP'** will disable the automatic patching of the DEBUG routines to allow you to see the program as it goes into the UniLab DEBUG routine. Remember to enable DEBUG again with **RSP** after you have used this diagnostic.

The trace display should show the breakpoint occuring at cycle # 0. The breakpoint should cause the program to go to the patched-in reserved memory area used by DEBUG. If you enter **RSTADR U.** the correct address will be displayed. If a single bit of the address disagrees with the correct one it probably indicates that that signal is not correctly connected to the target processor. Also if the CONTrol bit which distinguishes read from write cycles is wrong, DEBUG will not function properly.

The trace should also show the proper execution of the breakpoint routine, which saves interrupt status, disables interrupts, possibly re-enables interrupts, then returns to the instruction after the breakpoint. Note the read cycles after the return instruction since they determine where the program will return to. You should see the same data that was pushed at the breakpoint being read back. If the stack pointer is not pointing to good working memory this return won't work properly causing DEBUG malfunction.

8051 Disassembler/DEBUG Glossary

=A value ---

Change A register to value.

=DPTR value ---

Change DPTR to 16 bit value.

=IE value ---

Change IE register to value.

=OVERLAY address--

Set the DEBUG overlay area. This area must be in emulation rom.

=PSW value ---

Change PSW register to value.

=R0 value ---

Change R0 register to value.

=R1 value ---

Change R1 register to value.

=R2 value ---

Change R2 register to value.

=R3 value ---

Change R3 register to value.

=R4 value ---

Change R4 register to value.

=R5 value ---

Change R5 register to value.

=R6 value ---

Change R6 register to value.

=R7 value ---

Change R7 register to value.

ASM adr ---

Assemble into memory at adr. When **ASM** is invoked, it sets the origin at adr. If adr is omitted, it will continue from the previous location used by **ASM-FILE**, **ASM** or **ORG**.

ASM-FILE adr start-scr# end-scr# ---

Assemble into memory at adr beginning from start-scr# and continuing to end-scr# . When **ASM-FILE** is invoked, it sets the origin at adr. If adr is omitted, it will continue from the previous address used by **ASM-FILE**, **ASM** or **ORG**.

BPEX --- user-macro

Word to patch in user word macro into breakpoint display. The user-macro will be executed each time DEBUG display appears.

BPEX2 --- user-macro2

Word to patch in user word macro #2 into breakpoint display.

BPINT (no parameters)

Initialization routine for setting up DEBUG vectors and saving overlay area.

BPTRIGADR --- adr

Address of software breakpoint vector. Can be displayed by typing **BPTRIGADR U.**

DM adr n ---

Disassemble n lines from adr.

DR? adr ---

Display contents of internal ram at adr. (adr = 0 to $FF, 80 to $FF not present in 8031 chip). Mainly useful for 8032 since **R!** will function identically on the range $00 to $7F.

DR! byte adr ---

Store byte to internal ram at adr. (adr = 0 to $FF, 80 to $FF not present in 8031 chip). Mainly useful for 8032 since **R!** will function identically on the range $00 to $7F.

EXEC opcode ---

Execute opcode from current breakpoint. This allows the user to execute a one byte opcode once DEBUG control is established.

FETCH (no parameters)

Macro definition setting up CONTrol values for fetch cycles. Not available in piggyback version

G adr ---

Set program counter to adr, release DEBUG control, let target resume execution.

GB adr1 adr2 ---

Set program counter to adr1, continue program execution, break at adr2 with display of registers.

HO (no parameters)

Display help screen for 8051 disassembler/DEBUG.

LTARG (no parameters)

Load sample target program, set up default MSEG, and default areas for emulation memory. This program is included to help the user in setting up the target system. The display of the **LTARG** program should help in analyzing difficulties when running the disassembler/DEBUG operating on the target system the first time.

LP (no parameters)

Go around a loop one time. You must have DEBUG control at a location within a loop that is going to execute at least one more time. This command should not be executed at the last instruction in the loop.

N (no parameters)

Execute next step in target program. This will not follow branches or jumps. Use of this command will clear any breakpoints set be **SMBP**.

NMI (no parameters)

Establish immediate DEBUG control via the NMI connection to the target system. Usually assigned to function key **F4**. After DEBUG control is established, this command will cause the processor to single step, following jumps and branches.

NMIVEC (no parameters)

Enable NMI features by automatically patching vector into **INT0** or **INT1** whenever target system is reset and analyzer is started. This is the default condition. Type **NMIVEC'** to disable this feature. **NM** and **RI/SI** will not be functional if this is disabled.

R? n ---

Display contents of Special Function Register n. (n = $80 to $FF). Will also work on internal ram from $00 to $7F.

R! byte n ---

Store byte to Special Function Register n. (n = $80 to $FF). Will also work on internal ram from $00 to $7F.

R (no parameters)

Display all registers at current breakpoint.

RB adr ---

Set breakpoint at adr, and gain DEBUG control when target address is reached. Restore any previous breakpoint (except those set by **SMBP**). Display all registers, flags, and next program step to be executed. DEBUG control can only be established by entering this command, the first breakpoint must be in emulation memory, and the **RB** command must be used with **RESET** enabled as in **RESET adr RB**.

READ (no parameters)

Macro definition setting up CONTrol values for read cycles. Not available for piggyback version

RI (no parameters)

Set up following trigger spec for **SI**. Usually used before trigger spec to use IRQ when conditons are met. **SI** starts analyzer and when conditions occur, a breakpoint is set, the target system is put into an idle loop, then DEBUG gains control.

RNAME (n ---)

Used to set the register name of register number n in the DEBUG display. Use is as follows:

n RNAME USER-NAME

This sets register n to show USER-NAME in DEBUG display rather than Rn.

RSP (no parameters)

Set up automatic patching of DEBUG vectors when target system is reset and the UniLab analyzer is started. The vectors are disabled by typing **RSP'**.

RSTADR --- adr

Address used by DEBUG for reserved and overlay area. Can be displayed by typing **RSTADR U.**

RZ (no parameters)

Release DEBUG control, resume target program at the address of the current program counter.

SI (no parameters)

Start the analyzer, send IRQ to target when current trigger spec is met and get DEBUG control. Normally used with **RI**, and a trigger spec.

Example:

RI 40 DATA 20 ADR ALSO 1234 ADR SI

TRAM (no parameters)

Change mode to allow UniLab memory examine and modify words (**MDUMP**, **M?**,**MM?**,**M!**, **MFILL**, etc.) to act on the 8051 emulation memory area. This is the default mode. It is only useful only if the target system has external ram at the same addresses as emulation rom.

TRAM' (no parameters)

Use memory examine/patch words on emulation memory. This is the default mode and should normally be in effect before setting a breakpoint, or setting an address for a DEBUG function (for **G**, **GW**, etc.) This command is only needed if these external ram addresses are the same as emulated memory. Otherwise, DEBUG will always show you emulated memory if the range requested is currently enabled.

WRITE (no parameters)

Macro definition setting up CONTrol values for write cycles. Not available in piggyback version